APPENDIX I

Chameleon Systems UMTS Rake Receiver Mapping Analysis

27-April-2000

Revision 0.41

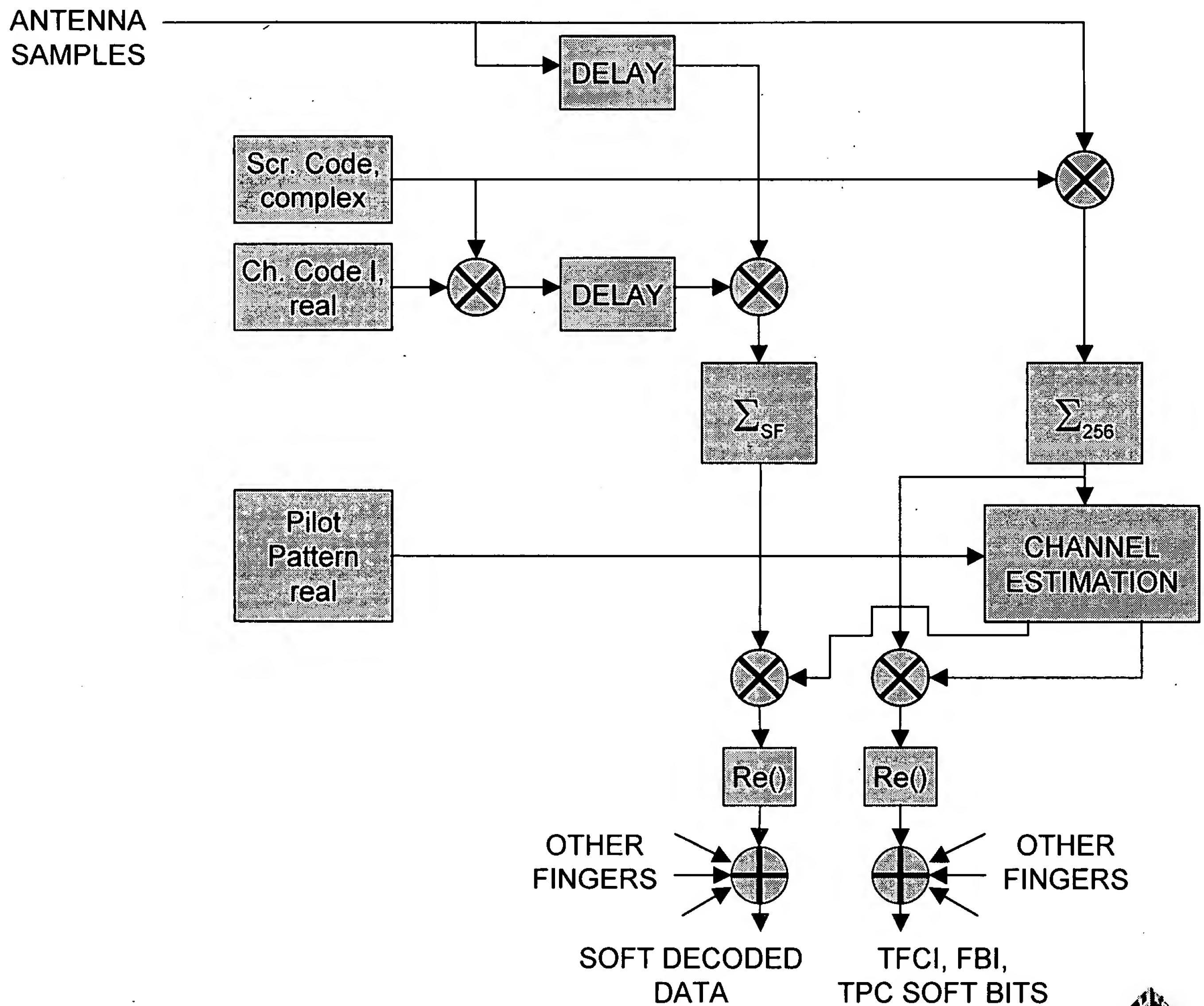
Dan Pugh & Mark Rollins
Chameleon Systems, Inc.
dan@chameleonsystems.com
rollins@chameleonsystems.com

CHAMELEON SYSTEMS, INC.

Rake Receiver
Requirements and Assumptions

- Requirements
  - Implement 32 users Pilot Processing and Data Despreading
    - Strive for target of 32 users in CS2112 (125 MHz)
  - Exceed target of 75 users in CS2112X (250 MHz)
- Assumptions
  - Maximum of 12 antennas
  - Maximum of 8 fingers per user
  - Average of 4 fingers per user
  - Spreading factors of 4 to 256 on DPDCH
  - Dual-port RAM at the input; Input order may be specified
  - ARC supplies scrambling code seeds
  - The Chameleon Processor is running at exactly 32 x the chip rate
  - An external Phase-locked-loop generates the 32 x (122.88) processor clock and is locked to the 3.84 MHz chip clock

Rake Processor Functional Block Diagram

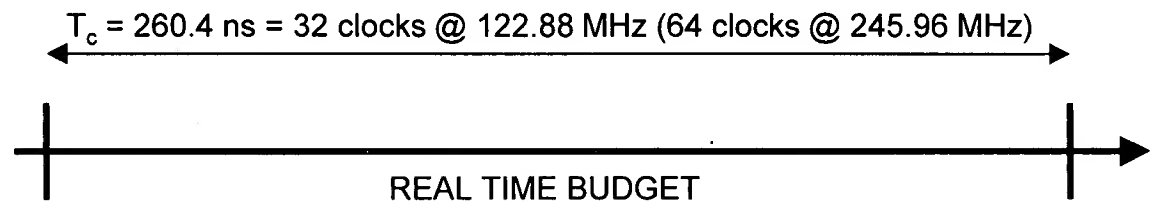


Rake Receiver
Description of Pipelined Operation

- Store a window of 128 chip samples sampled $T_c/2$ apart
- In each grouping of 256 consecutive 122.88 MHz clocks:

  At EVERY 122.88 MHz clock period:
  - Read 8 consecutive chip samples at the correct multi-path delay offset
  - Align the eight samples to the Despreading Code (Gold Code)
  - Multiply the 8 data samples by the 8 appropriate Despreading Codes
  - Sum the eight despread chips into one sample (two samples if SF=4)
  - Accumulate the sum-of-eight chips into a single despread symbol
- Send the despread Pilot Symbols to the ARC processor
  - Calculate Channel Estimation Weights
  - Multiply TFCI, FBI, TPC bits by Channel Estimation Weights
  - Sum up to six fingers to form each soft bit (symbol)

Fundamental Scheduling Analysis

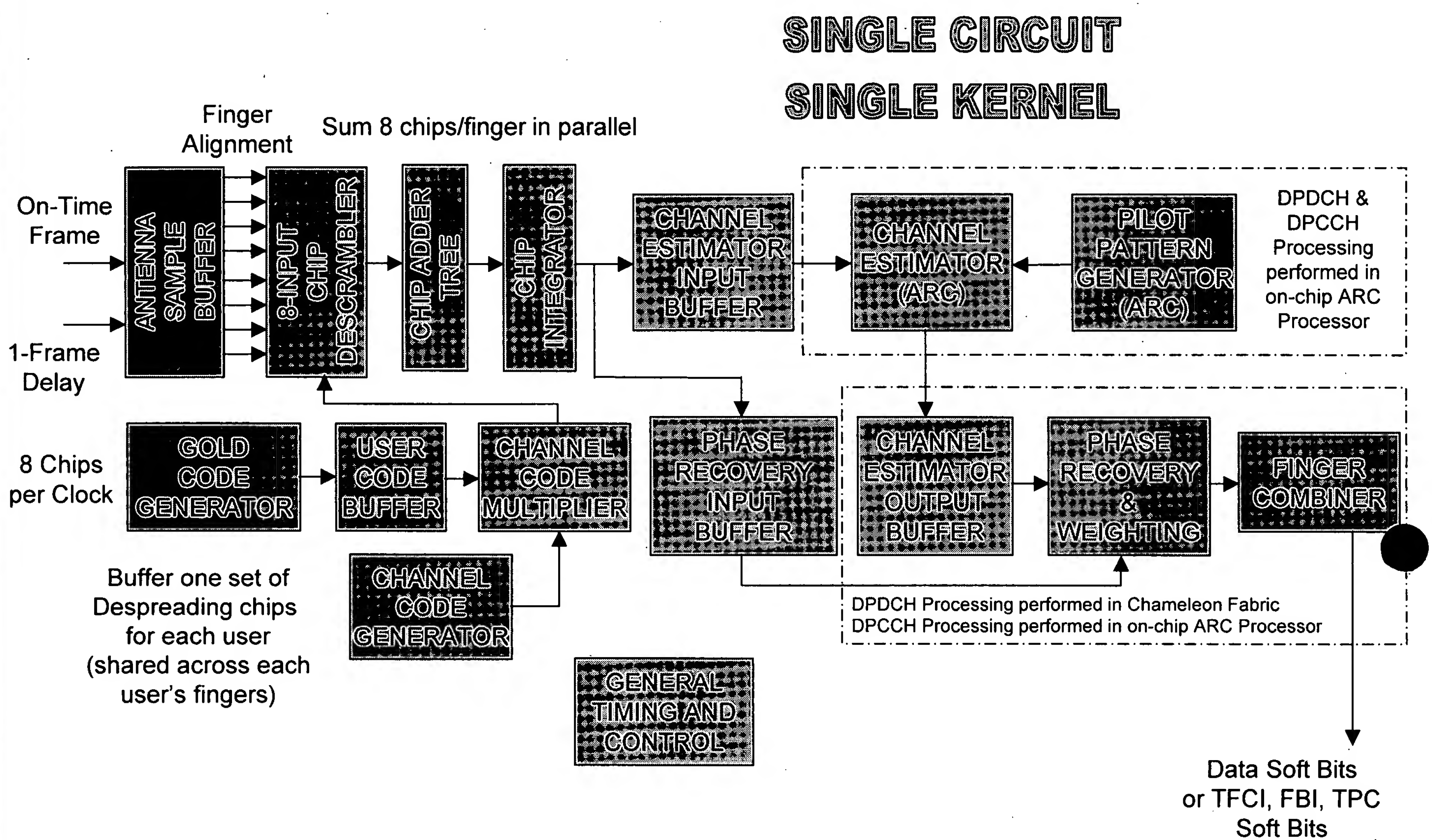


Parallel Implementation:

- Each circuit processes 8 chips per clock
- Throughput of 256 fingers per circuit
- Populate device with single circuit
- Achieves 128 pilot data fingers (32 users) @ 122.88 MHz
- Achieves 128 data fingers (same 32 users) @ 122.88 MHz
- Achieves 512 fingers (64 users) @ 245.96 MHz
- Utilizes single centralized control unit
- Fits within a single Chameleon device

Parallel Implementation is more efficient than multiple instantiations of individual units

Chameleon Rake Processor High-Level Partitioning Overview



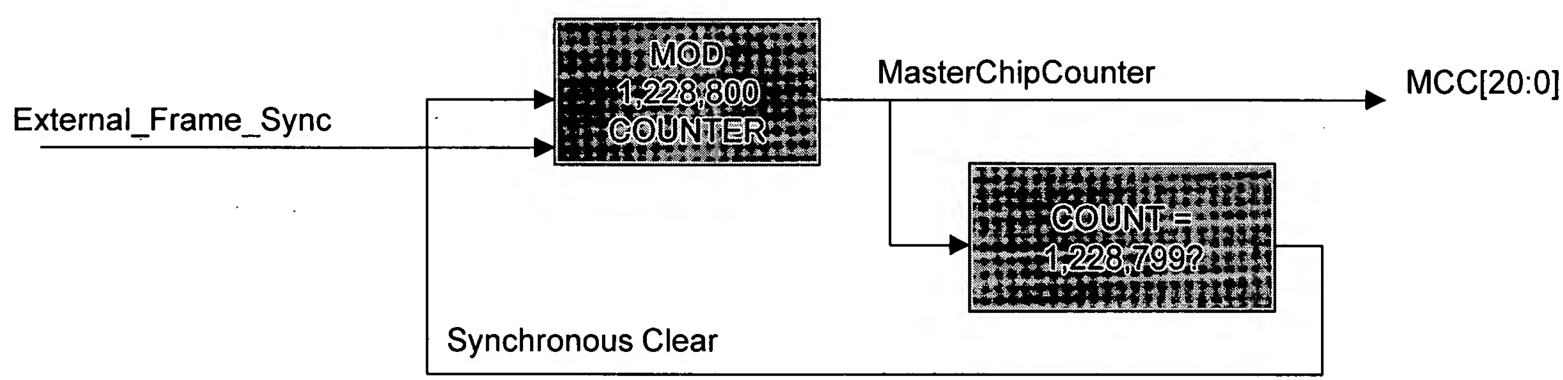
CHAMELEON SYSTEMS, INC.

General Timing and Control

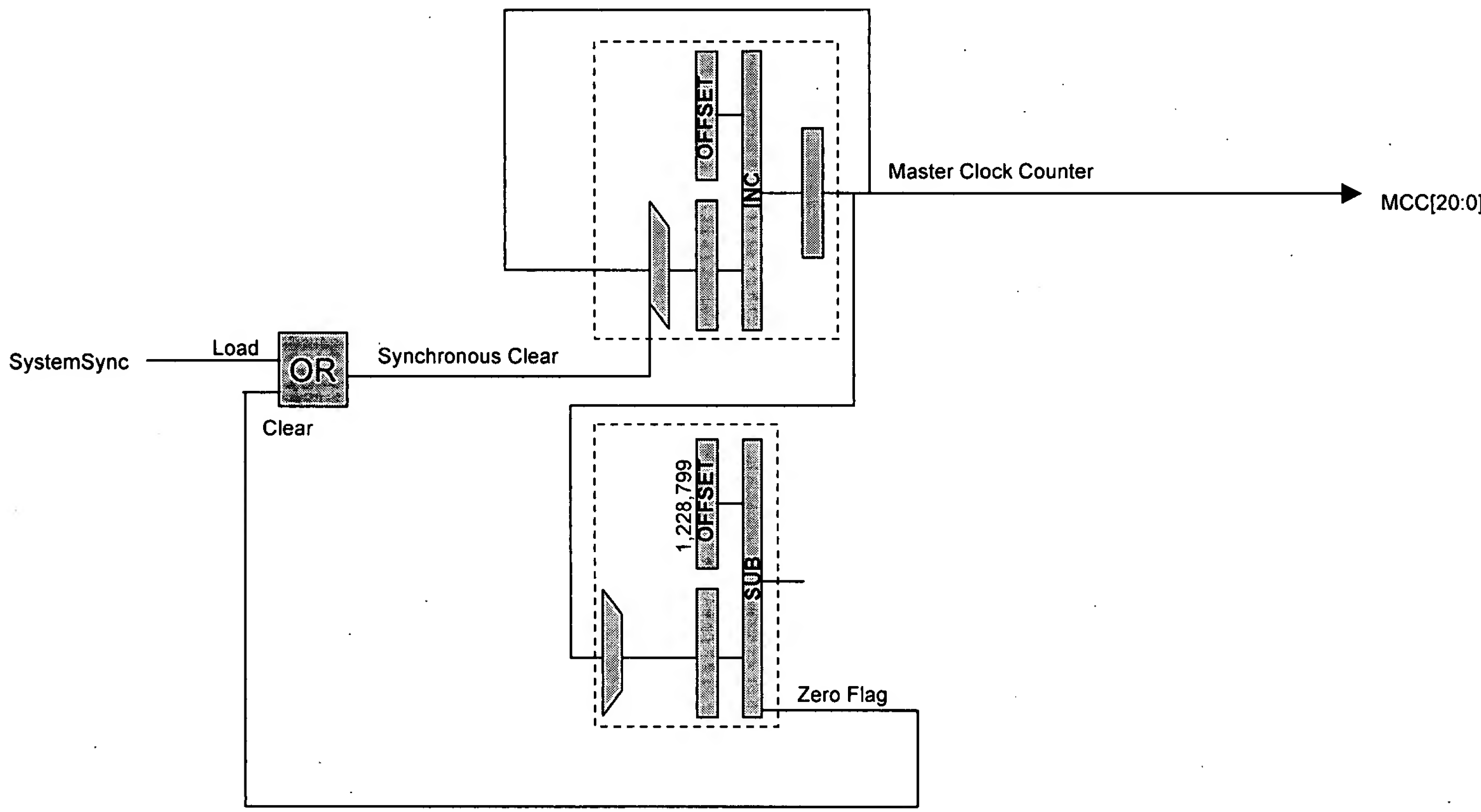
- The Rake Processor is synchronized to the chip-rate clock with an external Phase-Locked-Loop (PLL)
- The Rake Processor clock runs at exactly 32 times the chip-rate clock (32 x 3.84 MHz = 122.88 MHz)
- All processes in the Rake Processor are synchronized to the Master Chip Counter (MCC)
- The MCC counts modulo 1,228,800 (32 clocks/chip x 15 slots x 2560 chips/slot) and is reset by the external Frame Sync signal
- Various bits in the MCC are used to generate the Antenna Sample Buffer Write Address

Master Chip Counter Block Diagram

The External_Frame_Sync signal clears the modulo 1,228,800 counter



Master Chip Counter Implementation



Master Chip Counter Resource Requirements

- Implementation of:

  32 Users @ 125 MHz

  48 Users @ 187.5 MHz

  64 Users @ 250 MHz

  - 2 DPUs
  - 0 LSMs

Antenna Sample Buffer Requirements and Assumptions

- **Requirements**
  - Provide memory in the Antenna Buffer so that a128-chip (256 half-chips) range of samples can be stored for all 12 antennas
  - This allows the Finger Alignment Buffer and the Antenna Buffer to be merged into a single memory
- **Assumptions**
  - The samples must be organized such that any eight samples T_c apart from one antenna may be read simultaneously from the buffer
  - Each Rake Receiver circuit (Chameleon chip) processor services from one to six sectors
  - Each Sector must process the primary and diversity antennas for the primary sector and the two adjacent sectors, for a minimum of six antennas per sector of coverage
  - Support of all 12 antennas is required

Antenna Sample Buffer General Description

- There are two partitions of the Antenna Sample Buffer
  - The first partition contains the on-time antenna data
  - The second partition contains the on-time antenna data that is delayed by approximately one frame
- The value of the delay is one frame plus the time required to compute the Spreading Factor from the TFCI bits
- The delayed data is required because the TFCI bits are in the same frame as the data that it controls

Antenna Sample Buffer General Description

- The Antenna Sample Buffer is used to store a large enough window of data that all six fingers may be accessed from any of the 12 antennas
- Data for each finger is read from the Antenna Buffer at the offset specified by the Path Searcher
- The Antenna Buffer is organized so that ANY eight consecutive samples T_c apart may be accessed from the buffer in a single clock cycle

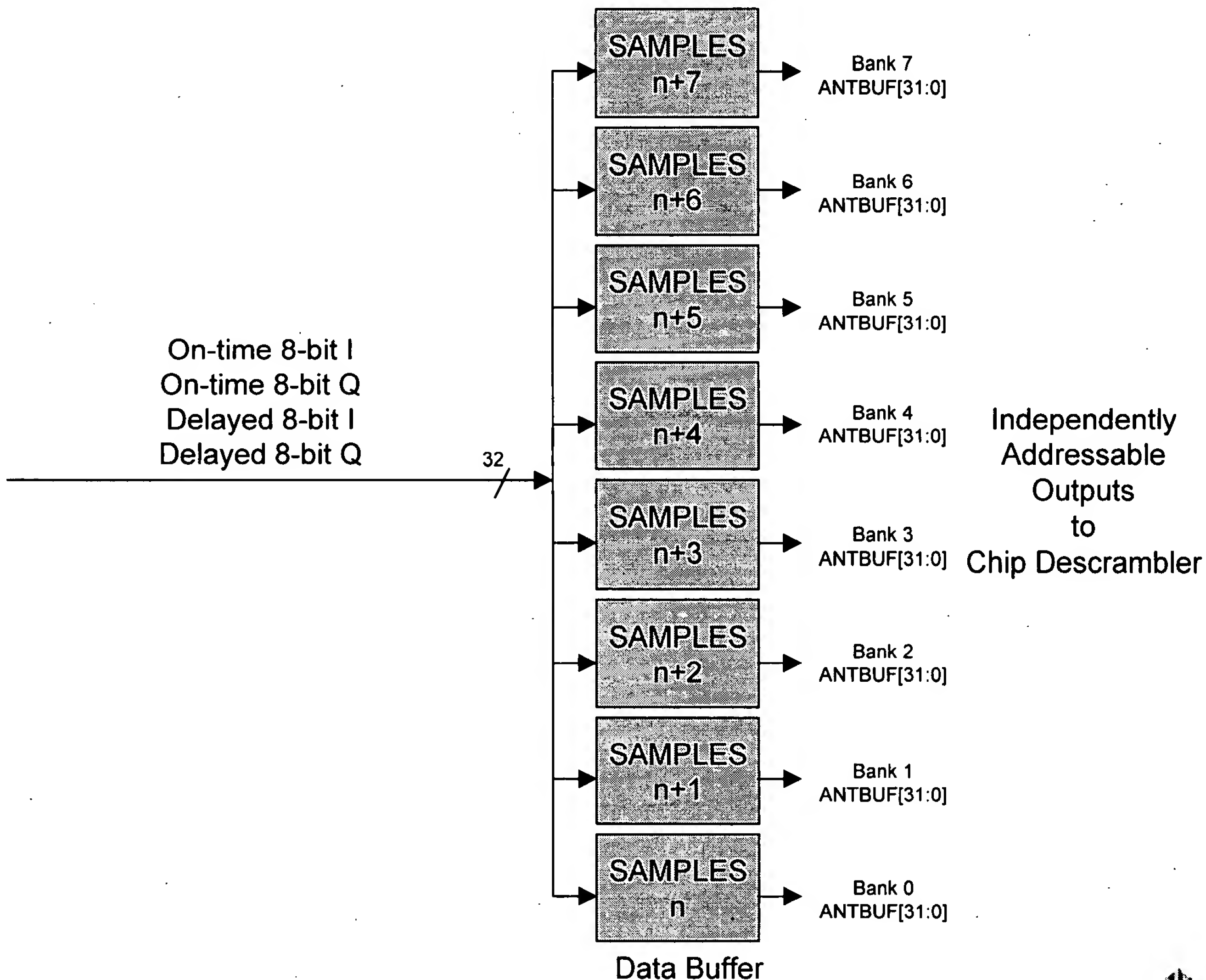
Antenna Sample Buffer Multipath Support Capabilities

- Specifications:
  - Chip-rate = 3.84 MHz (260.4 ns, wavelength = 78.125 m)
  - Maximum distance of mobile user to base station = 7000 m
- The 7000 m user radius corresponds 0-89.6 chips of line-of-sight delay
- The 128-chip Antenna Sample Buffer provides an additional 38.4-chip buffer to support multipath components
- This corresponds to support for multipath components with up to 10,000 ns (3000 m) delay for all 12 antennas
- If the Antenna Sample Buffer is retasked to support only 8 antennas (two adjacent sectors) the maximum multipath support may be increased to 26,664 ns (8000 m)

Antenna Sample Buffer Multipath Support Capabilities

- Given a 5000m user radius (3G TS25.942)
  - A 5000 m user radius corresponds 64 chips of line-of-sight delay
  - The 128-chip Antenna Sample Buffer provides an additional 64-chip buffer to support multipath components
- This corresponds to 10,000m of support for a combination of the user radius plus multipath delay with all 12 antennas
- The Antenna Sample Buffer may be retasked to support:
  - 8 antennas (two sectors) for 15,000m of user radius/multipath support
  - 6 antennas (one sector) for 20,000m of user radius/multipath support

Antenna Sample Buffer Functional Block Diagram



Antenna Sample Buffer Memory Map for Bank 0

ADDRESS	ANTBUF[31:0]
0x5FC	Antenna 11, Sample 120, HalfChip 1
0x5F8	Antenna 11, Sample 120, HalfChip 0
0x5F4	Antenna 11, Sample 112, HalfChip 1
0x5F0	Antenna 11, Sample 112, HalfChip 0
⋮	⋮
0x094	Antenna 1, Sample 16, HalfChip 1
0x090	Antenna 1, Sample 16, HalfChip 0
0x08C	Antenna 1, Sample 8, HalfChip 1
0x088	Antenna 1, Sample 8, HalfChip 0
0x084	Antenna 1, Sample 0, HalfChip 1
0x080	Antenna 1, Sample 0, HalfChip 0
0x07C	Antenna 0, Sample 120, HalfChip 1
0x078	Antenna 0, Sample 120, HalfChip 0
0x074	Antenna 0, Sample 112, HalfChip 1
0x070	Antenna 0, Sample 112, HalfChip 0
⋮	⋮
0x014	Antenna 0, Sample 16, HalfChip 1
0x010	Antenna 0, Sample 16, HalfChip 0
0x00C	Antenna 0, Sample 8, HalfChip 1
0x008	Antenna 0, Sample 8, HalfChip 0
0x004	Antenna 0, Sample 0, HalfChip 1
0x000	Antenna 0, Sample 0, HalfChip 0

ANTENNA SAMPLE BUFFER WORD CONTENTS

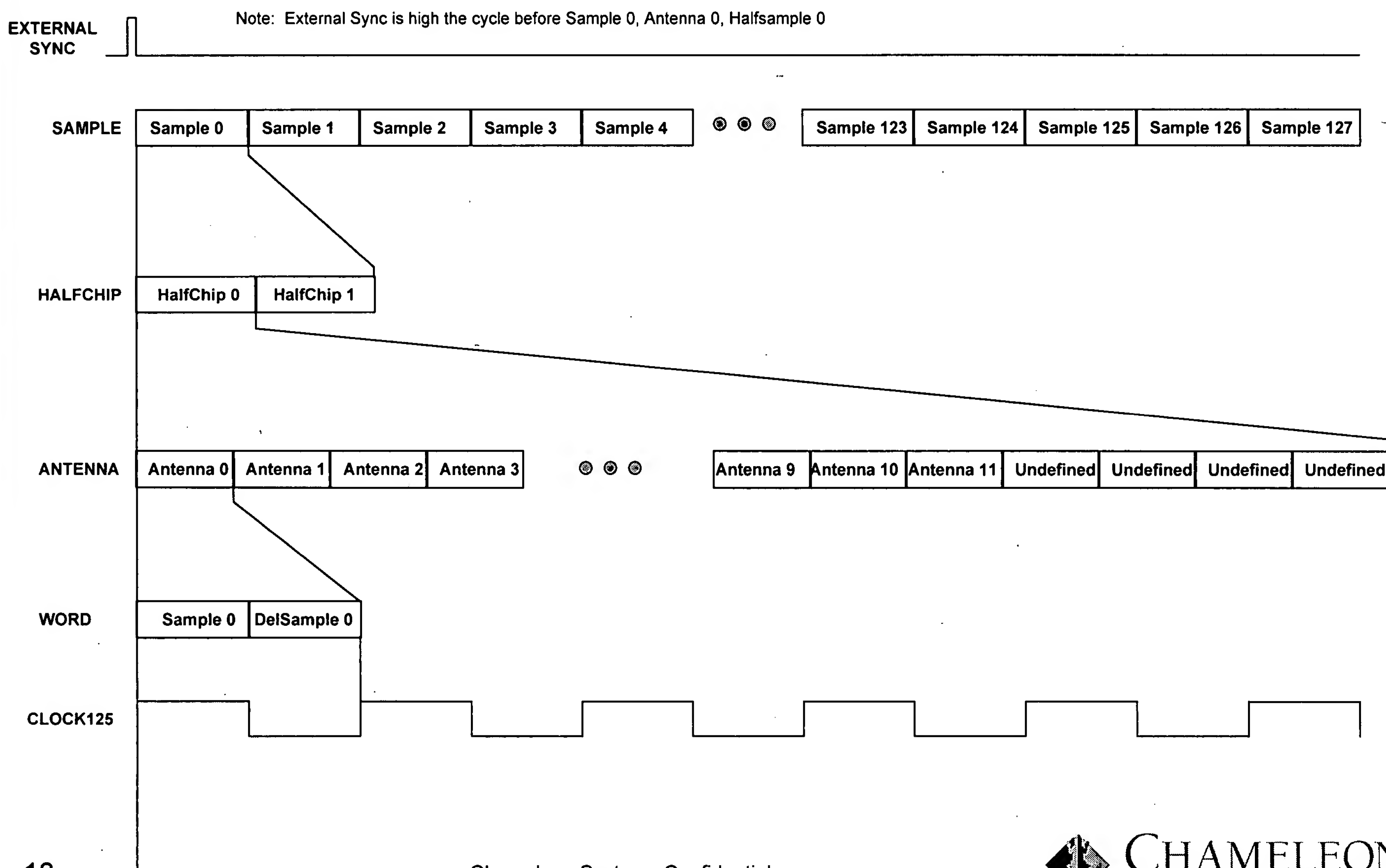
ANTBUF[31:24]	ANTBUF[23:16]	ANTBUF[15:8]	ANTBUF[7:0]
Non-Delayed Q[7:0]	Delayed Q[7:0]	Non-Delayed I[7:0]	Delayed I[7:0]

Note:

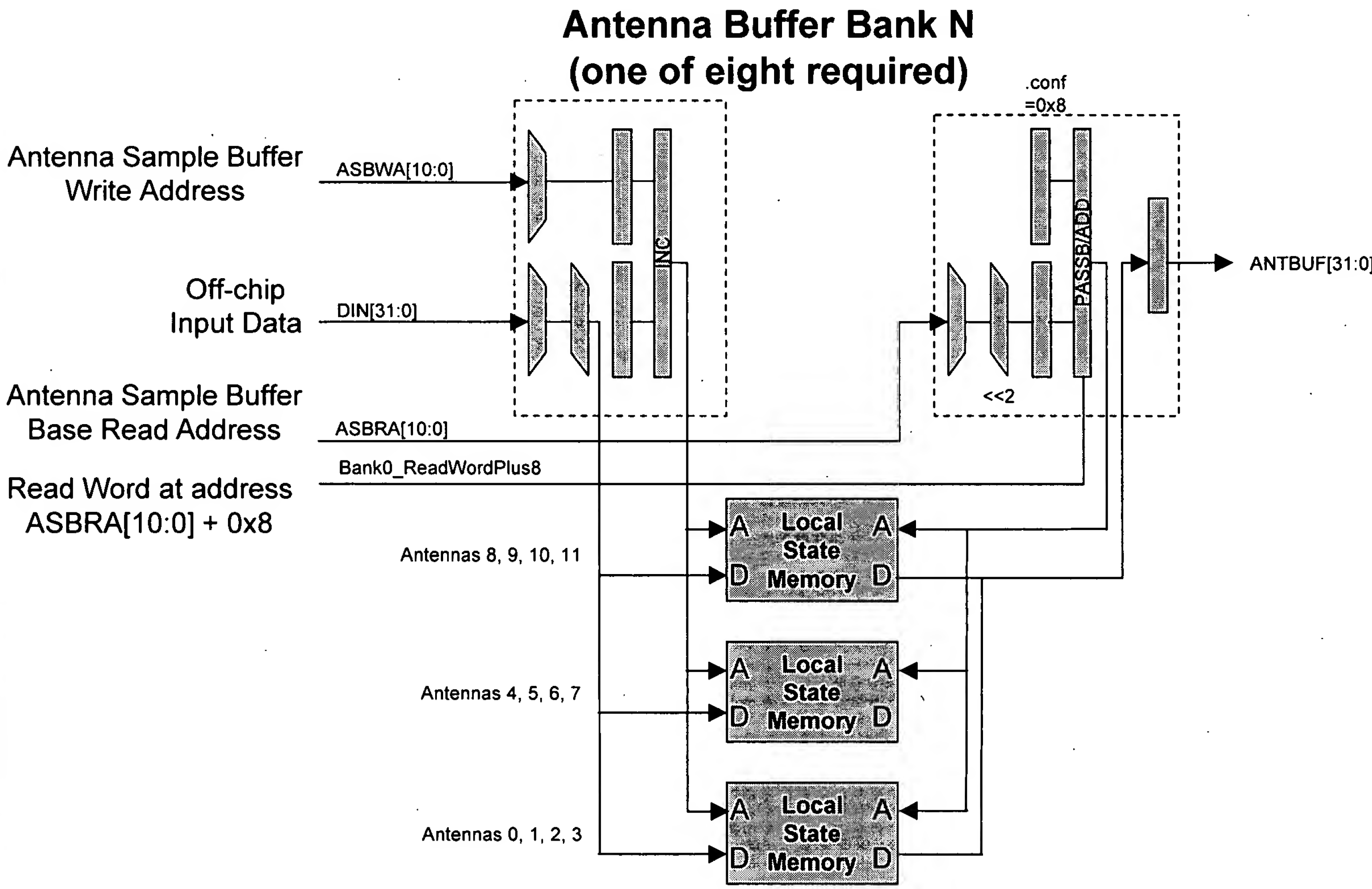
Sample n HalfChip 0 is used to denote the sample at time n

Sample n HalfChip 1 is used to denote the sample at time n+1/2

External Antenna Sample Buffer Bus Organization



Antenna Sample Buffer Data Buffer Implementation



Antenna Sample Buffer Write Address Bit Definitions

- The Modulo 1,228,00 Master Chip Counter (MCC) bit fields may be defined with respect to the input data samples to the Antenna Sample Buffer Write Address (ASBWA)
- The Antenna Sample Buffer Write Address Generator bits are MCC bits that have been reordered to properly store the input samples

MCC BIT	DESCRIPTION
MCC[11]	ChipCount[6]
MCC[10]	ChipCount[5]
MCC[9]	ChipCount[4]
MCC[8]	ChipCount[3]
MCC[7]	ChipCount[2]
MCC[6]	ChipCount[1]
MCC[5]	ChipCount[0]
MCC[4]	HalfChip
MCC[3]	Antenna[3]
MCC[2]	Antenna[2]
MCC[1]	Antenna[1]
MCC[0]	Antenna[0]

ANTENNA SAMPLE BUFFER BANK ADDRESS (for each of the eight banks)

ASBWA BIT	DESCRIPTION
ASBWA[10]	Antenna[3]
ASBWA[9]	Antenna[2]
ASBWA[8]	Antenna[1]
ASBWA[7]	Antenna[0]
ASBWA[6]	ChipCount[6]
ASBWA[5]	ChipCount[5]
ASBWA[4]	ChipCount[4]
ASBWA[3]	ChipCount[3]
ASBWA[2]	HalfChip
ASBWA[1]	0
ASBWA[0]	0

Note:
ChipCount[2:0] is used to select one of eight physical memory banks

The one-frame-delayed Samples are stored in bytes 0 and 2 of the Antenna Sample Buffer, while the non-delayed samples are stored in bytes 1 and 3 of the Antenna Sample Buffer

Each Bank requires 3 LSMs

Antenna Sample Buffer Write Address Generation

- If we look at the fields of the Write Address Generator we see that:
  - CASE A: When MCC[11:0]=0xFFF
    - ASBWA[10:7] = Antenna[3:0] must be cleared
    - ASBWA[6:3] = ChipCount[6:3] must be cleared
    - ASBWA[2] = HalfChip must be cleared
  - CASE B: When MCC[7:0]=0xFF
    - ASBWA[10:7] = Antenna[3:0] must be cleared (by incrementing by one)
    - ASBWA[6:3] = ChipCount[6:3] increments by one
    - ASBWA[2] = HalfChip must be cleared
  - CASE C: When MCC[4:0]=0x1F
    - ASBWA[10:7] = Antenna[3:0] increments by one
    - ASBWA[2] = HalfChip must be cleared
  - CASE D: When MCC[4:0]=0x0F
    - ASBWA[10:7] = Antenna[3:0] increments by one
    - ASBWA[2] = HalfChip must be set
  - CASE E: Otherwise
    - ASBWA[10:7] = Antenna[3:0] must be incremented by one

Antenna Sample Buffer Write Address Generation Implementation

- To implement the above five cases, let us define two registers:
  - REGA = 128 - 4 = 124 = 0x7C
  - REGB = 128 = 0x80
- CASE A: This state occurs when the entire Antenna Sample Buffer has been written and the address field must be cleared so that the buffer may start again at the beginning address
- MCC[11:0]=0xFFF
  - ASBWA[10:7] = Antenna[3:0] must be cleared
  - ASBWA[6:3] = ChipCount[6:3] must be cleared
  - ASBWA[2] = HalfChip must be cleared
    - ALUB = shifterconst=0x0
    - ALU = PASSB

Antenna Sample Buffer Write Address Generation Implementation

- CASE B: This state occurs when both a chip and HalfChip have been written to each of the eight banks and the address field must point back to the first bank
- MCC[7:0]=0xFF
  - ASBWA[10:7] = Antenna[3:0] increments by one
  - ASBWA[6:3] = ChipCount[6:3] increments by one
  - ASBWA[2] = HalfChip must be cleared
    - Since we know ChipCount[6:3] ≠ 0xF, incrementing ChipCount[6:3] will not generate a carry into Antenna[3:0]
    - Since we know HalfChip=1, we can increment HalfChip and it will toggle HalfChip AND generate a carry to increment ChipCount[6:3]
    - ALUA = ALUOUTREG
    - ALUB = 128 + 4 = REGB OR shifterconst=0x4
    - ALU = ALUA + ALUB

Antenna Sample Buffer Write Address Generation Implementation

- CASE C: This state occurs when after a sample (HalfChip=1) has been written to a single bank for all twelve antennas and the address field must point to the next memory bank
- MCC[4:0]=0x1F
  - ASBWA[10:7] = Antenna[3:0] increments by one
  - ASBWA[2] = HalfChip must be cleared
    - ALUA = 128 - 4 = REGA
    - ALUB = ALUOUTREG
    - ALU = ALUA + ALUB

Antenna Sample Buffer
Write Address Generation Implementation

- CASE D: This state occurs when after a sample (HalfChip=0) has been written to a single bank for all twelve antennas and the address field must point to the next half-chip address within the same memory bank
- MCC[4:0]=0x0F
  - ASBWA[10:7] = Antenna[3:0] increments by one
  - ASBWA[2] = HalfChip must be set
    - ALUA = ALUOUTREG
    - ALUB = 128 + 4 = REGB OR shifterconst=0x4
    - ALU = ALUA + ALUB

Antenna Sample Buffer
Write Address Generation Implementation

- CASE E: This state occurs when the conditions for any of the cases A through D are not met and the address field must point to the next antenna sample
  - ASBWA[10:7] = Antenna[3:0] must be incremented by one
    - ALUA = ALUOUTREG
    - ALUB = 128 = REGB
    - ALU = ALUA + ALUB

- Note that for both CASE B and CASE D, the instructions to the DPU are identical
- We therefore only have four unique states for the DPU

Antenna Sample Buffer Write Address Generation Implementation

- Let us define the following four states
  - State 00: DPU instruction for CASE A
  - State 01: DPU instruction for CASE B and CASE D
  - State 10: DPU instruction for CASE C
  - State 11: DPU instruction for CASE E
- Let us use the notation of A when case A is true and /A when CASE A is not active, and X represents don't care
- We also must assign priority to the five cases since the priority is base on the MCC[11:0] higher order bits having higher priority
  - Priority 1: A , CASE B=X, C=X, D=X E=X — State 00
  - Priority 2: (!A & B), C=X, D=X E=X — State 01
  - Priority 3: (!A & !B & C & !D) — State 10
  - Priority 3: (!A & !B & !C & D) — State 01
    - Note CASE C and CASE D have equal priority
  - Priority 4: !A & !B & !C & !D — State 11
    - Note that the encoding !A & !B & C & D is not physically possible

CHAMELEON SYSTEMS, INC.

Antenna Sample Buffer Write Address Generation Implementation

- Let Pn represent the priority for priority with level n
  We can fill in the priority assignments for the Karnaugh map entries for the sixteen possibilities for P0, P1, P2, and P3

PRIORITY ASSIGNMENTS

D=1 C=1

		00	01	11	10
	00	P4	P3	X	P3
B=1	01	P2	P2	P2	P2
	11	P1	P1	P1	P1
A=1	10	P1	P1	P1	P1

Antenna Sample Buffer
Write Address Generation Implementation

If we fill in the DPU state tables with the state assignments of the previous page:

Priority 1: A , CASE B=X, C=X, D=X E=X — State 00
Priority 2: (!A & B), C=X, D=X E=X — State 01
Priority 3: (!A & !B & C & !D) — State 10
Priority 3: (!A & !B & !C & D) — State 01
Note CASE C and CASE D have equal priority
Priority 4: !A & !B & !C & !D — State 11
Note that the encoding !A & !B & C & D is not physically possible

STATE[1] (columns: D=1 over 01, 11; C=1 over 11, 10; rows: B=1 over 01, 11; A=1 over 11, 10)

	00	01	11	10
00	1	0	X	1
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

STATE[0] (columns: D=1 over 01, 11; C=1 over 11, 10; rows: B=1 over 01, 11; A=1 over 11, 10)

	00	01	11	10
00	1	1	X	0
01	1	1	1	1
11	0	0	0	0
10	0	0	0	0

STATE[1] = !A & !B & !D
STATE[0] = (!A & B) | (!B & !C)

Note that we do not need to compute variable D

Antenna Sample Buffer Write Address Generation Implementation

In order to be able to decode the conditions A, B, C, and D and their inverses, without using excessive product terms, we decode a count one before the desired count and register the result

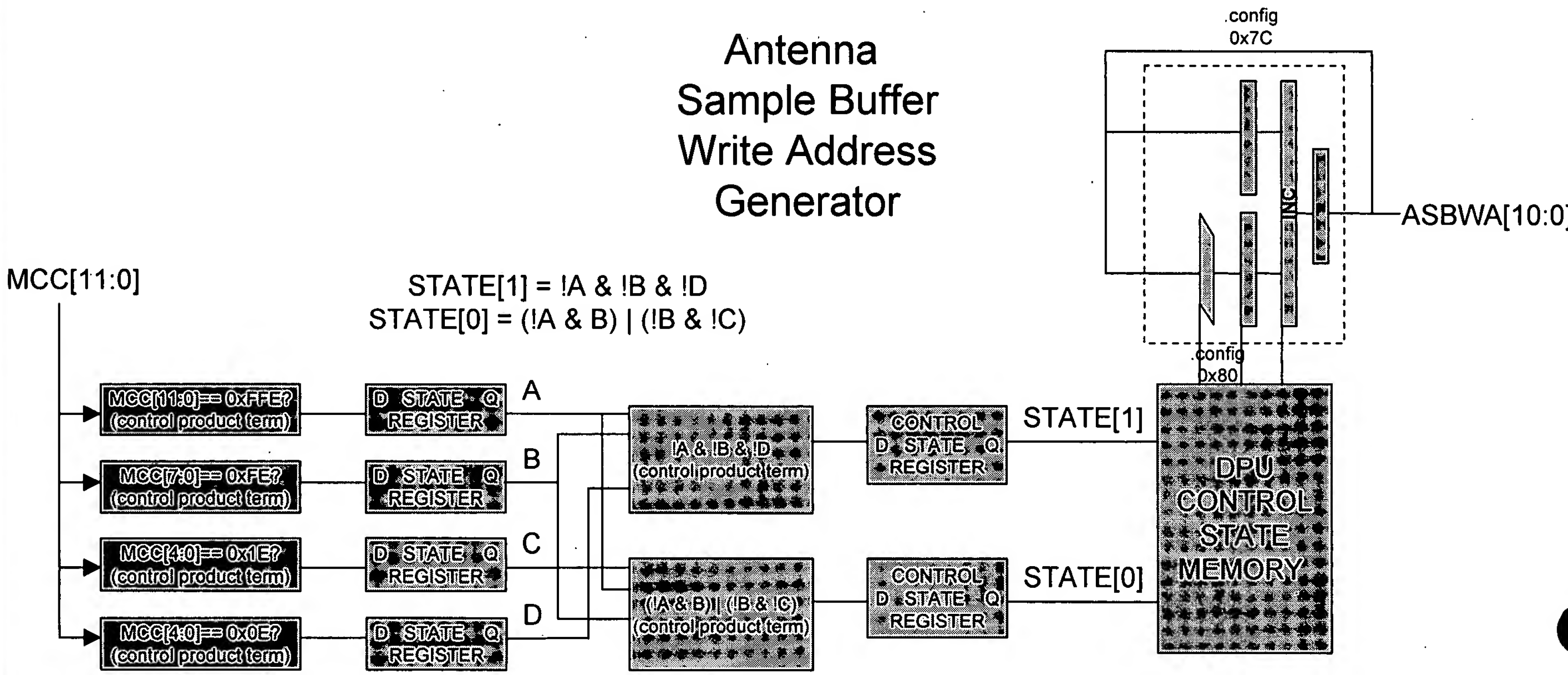
REGA: MCC[11:0] = 0xFFE = MCC[11] & MCC[10] & MCC[9] & MCC[8] & MCC[7] & MCC[6] & MCC[5] & MCC[4] & MCC[3] & MCC[2] & MCC[1] & !MCC[0]

REGB: MCC[7:0] = 0xFE = MCC[7] & MCC[6] & MCC[5] & MCC[4] & MCC[3] & MCC[2] & MCC[1] & !MCC[0]

REGC: MCC[4:0] = 0x1E = MCC[4] & MCC[3] & MCC[2] & MCC[1] & !MCC[0]

REGD: MCC[4:0] = 0x0E = !MCC[4] & MCC[3] & MCC[2] & MCC[1] & !MCC[0]

Antenna Sample Buffer Write Address Generator Implementation



Antenna Sample Buffer Bank Write Enable Generation

- Each of the eight banks in the Antenna Sample Buffer contains the samples eight chips apart for all twelve antennas.
- The control must generate a Antenna Sample Buffer Write Enable ($ASBWE_n$) signal for each of the n banks
- The timing for the $ASBWE_n$ signals is based upon the Master Chip Counter bits MCC[7:0]
- Note that the MCC counter implicitly counts through values for sixteen antennas, but the write enable signals are only enabled during the first twelve

Antenna Sample Buffer Bank Write Enable Generation

- ASBWE$_n$ is active when MCC[7:5] = ChipCount[2:0] matches the addressed memory bank and MCC[3:0] = Antenna[3:0] addresses antennas 0 to 11:
  - ASBWE$_0$ = !MCC[7] & !MCC[6] & !MCC[5] & !(MCC[3] & MCC[2])
    ASBWE$_1$ = !MCC[7] & !MCC[6] & MCC[5] & !(MCC[3] & MCC[2])
  - ASBWE$_2$ = !MCC[7] & MCC[6] & !MCC[5] & !(MCC[3] & MCC[2])
  - ASBWE$_3$ = !MCC[7] & MCC[6] & MCC[5] & !(MCC[3] & MCC[2])
  - ASBWE$_4$ = MCC[7] & !MCC[6] & !MCC[5] & !(MCC[3] & MCC[2])
  - ASBWE$_5$ = MCC[7] & !MCC[6] & MCC[5] & !(MCC[3] & MCC[2])
  - ASBWE$_6$ = MCC[7] & MCC[6] & !MCC[5] & !(MCC[3] & MCC[2])
  - ASBWE$_7$ = MCC[7] & MCC[6] & MCC[5] & !(MCC[3] & MCC[2])

CHAMELEON SYSTEMS, INC.

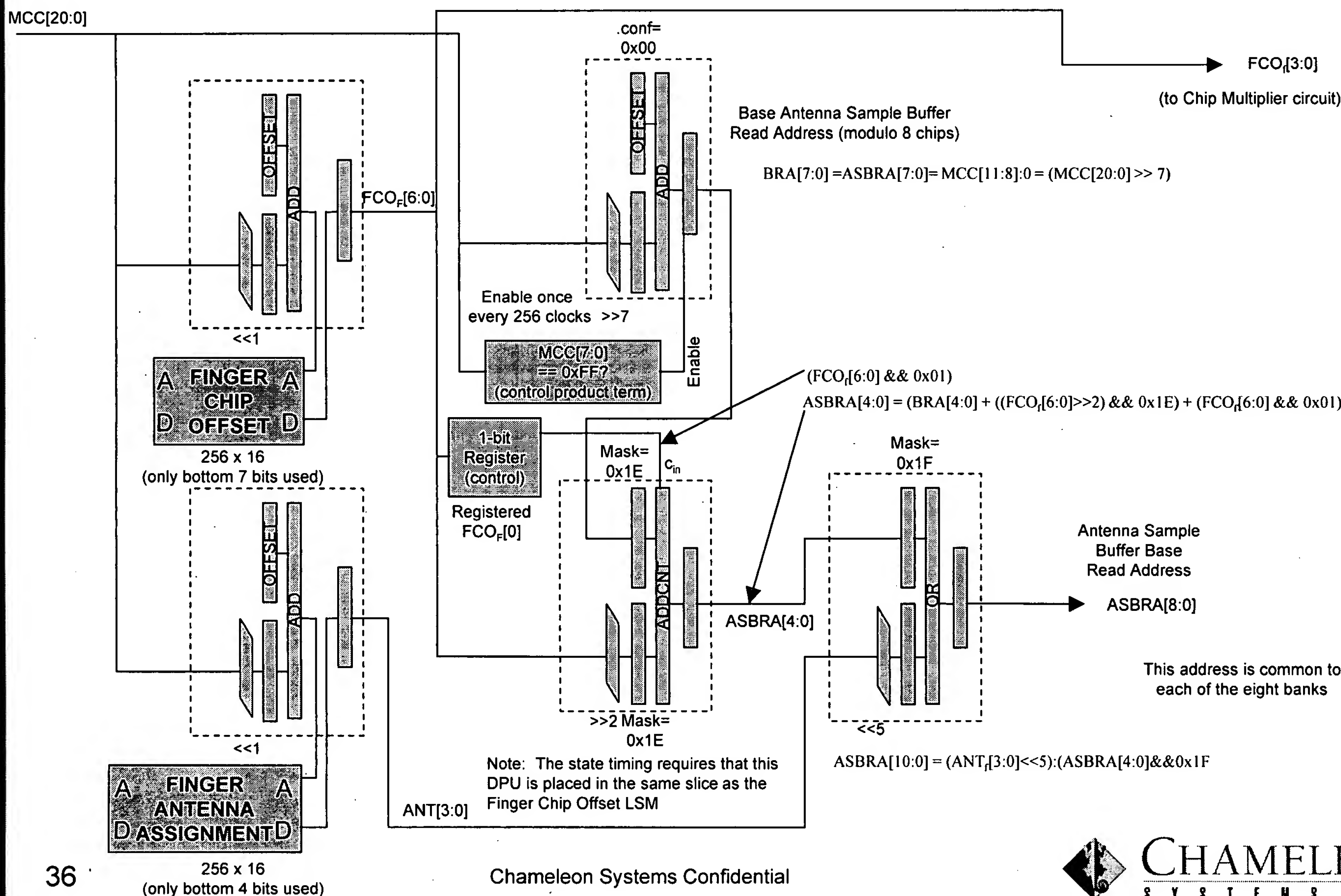
Antenna Sample Buffer Read Address Generation (1 of 2)

- Compute the Antenna Sample Buffer Base Read Address (ASBRA) for the Antenna Sample Buffer as follows:
  - The Base Read Address (BRA) is the fixed offset between the Antenna Sample Buffer Write Address and the Antenna Sample Buffer Read Address (ASRA) for zero finger chip offset
  - For each of the 256 fingers:
    - Add the Finger Chip Offset (FCO_f) to the BRA
    - Merge the Finger Antenna Assignment (ANT_f) bits to the ASBRA
    - The Antenna Sample Buffer Read Address (ASBA) is formed by shifting the ASBRA left two bit positions

Antenna Sample Buffer Read Address Generation (2 of 2)

- Compute the Base Read Address (BRA):
  - BRA[4:0] =ASBRA[4:0]= MCC[11:8]:0 = ((MCC[20:0] >> 7) && 0x1E)
    - The Base Read Address is latched to remain constant for 256 clocks
    - An offset can be added if necessary to adjust timing
- For each of the f fingers (0-255)
  - Read the antenna assignment (ANT_f) for the current finger
  - Read the Finger Chip Offset (FCO_f) for the current finger
  - Add the Finger Chip Offset to the Base Address Register
    - ASBRA[4:0] = (BRA[4:0] + ((FCO_f[6:0]>>2) && 0x1E) + (FCO_f[6:0] && 0x01)
  - The Antenna Assignment ANT_f[3:0] is placed in bits ASBRA[8:5]
    - ASBRA[8:0] = (ANT_f[3:0]<<5):ASBRA[4:0]
  - The Antenna Sample Buffer Read Address (ASRA) is formed by shifting ASBRA left two bit positions
    - ASRA[10:0] = ASBRA[8:0] << 2

Antenna Sample Buffer Base Read Address Generator Implementation



Antenna Sample Buffer Read Address Offset Circuit

- The Antenna Sample Buffer Base Base Read Address, (BRA) is on a multiple of eight sample boundary
- If the Finger Chip Offset for a finger, FCO_f, is not on a multiple-of-eight sample boundary, then the correct eight consecutive samples are not output on the memory
- To output the correct words, some of the addresses must not select the word specified by the ASBRA, but the next full sample into the buffer
- $FCO_f[3:1]$ are used to determine which of the eight Antenna Sample Buffer address generators need to have their Bankn_ReadWordPlus8 asserted in order to read the next eighth sample into the buffer

Antenna Sample Buffer Read Address Offset Circuit

- The ASBRA calculation requires that the Finger Chip Offset for a given finger, FCO_f, is read out of the FCO memory before it is needed for the Read Address Offset Circuit
  - $FCO_f[0]$ is needed one cycle after it is read from the FCO memory, so the DPU using $FCO_f[0]$ as a control input must be placed in the same tile as the FCO memory, because $FCO_f[0]$ is delayed one cycle through the Control State Memory
  - $FCO_f[3:1]$ is needed by the Read Address Offset Circuit two cycles after it is read from the FCO memory
  - An additional delay of one cycle for $FCO_f[3:1]$ is achieved if $FCO_f[3:1]$ is routed through Broadcast Bit horizontal long lines and the following equations are used:

Antenna Sample Buffer Bankn_ReadWordPlus8 Generation

```
Case FCOf[3:1]
  0: Bank0_ReadWordPlus8 = 0
     Bank1_ReadWordPlus8 = 0
     Bank2_ReadWordPlus8 = 0
     Bank3_ReadWordPlus8 = 0
     Bank4_ReadWordPlus8 = 0
     Bank5_ReadWordPlus8 = 0
     Bank6_ReadWordPlus8 = 0
     Bank7_ReadWordPlus8 = 0

  1: Bank0_ReadWordPlus8 = 1
     Bank1_ReadWordPlus8 = 0
     Bank2_ReadWordPlus8 = 0
     Bank3_ReadWordPlus8 = 0
     Bank4_ReadWordPlus8 = 0
     Bank5_ReadWordPlus8 = 0
     Bank6_ReadWordPlus8 = 0
     Bank7_ReadWordPlus8 = 0

  2: Bank0_ReadWordPlus8 = 1
     Bank1_ReadWordPlus8 = 1
     Bank2_ReadWordPlus8 = 0
     Bank3_ReadWordPlus8 = 0
     Bank4_ReadWordPlus8 = 0
     Bank5_ReadWordPlus8 = 0
     Bank6_ReadWordPlus8 = 0
     Bank7_ReadWordPlus8 = 0

  3: Bank0_ReadWordPlus8 = 1
     Bank1_ReadWordPlus8 = 1
     Bank2_ReadWordPlus8 = 1
     Bank3_ReadWordPlus8 = 0
     Bank4_ReadWordPlus8 = 0
     Bank5_ReadWordPlus8 = 0
     Bank6_ReadWordPlus8 = 0
     Bank7_ReadWordPlus8 = 0

  4: Bank0_ReadWordPlus8 = 1
     Bank1_ReadWordPlus8 = 1
     Bank2_ReadWordPlus8 = 1
     Bank3_ReadWordPlus8 = 1
     Bank4_ReadWordPlus8 = 0
     Bank5_ReadWordPlus8 = 0
     Bank6_ReadWordPlus8 = 0
     Bank7_ReadWordPlus8 = 0

  5: Bank0_ReadWordPlus8 = 1
     Bank1_ReadWordPlus8 = 1
     Bank2_ReadWordPlus8 = 1
     Bank3_ReadWordPlus8 = 1
     Bank4_ReadWordPlus8 = 1
     Bank5_ReadWordPlus8 = 0
     Bank6_ReadWordPlus8 = 0
     Bank7_ReadWordPlus8 = 0

  6: Bank0_ReadWordPlus8 = 1
     Bank1_ReadWordPlus8 = 1
     Bank2_ReadWordPlus8 = 1
     Bank3_ReadWordPlus8 = 1
     Bank4_ReadWordPlus8 = 1
     Bank5_ReadWordPlus8 = 1
     Bank6_ReadWordPlus8 = 0
     Bank7_ReadWordPlus8 = 0

  7: Bank0_ReadWordPlus8 = 1
     Bank1_ReadWordPlus8 = 1
     Bank2_ReadWordPlus8 = 1
     Bank3_ReadWordPlus8 = 1
     Bank4_ReadWordPlus8 = 1
     Bank5_ReadWordPlus8 = 1
     Bank6_ReadWordPlus8 = 1
     Bank7_ReadWordPlus8 = 0
```

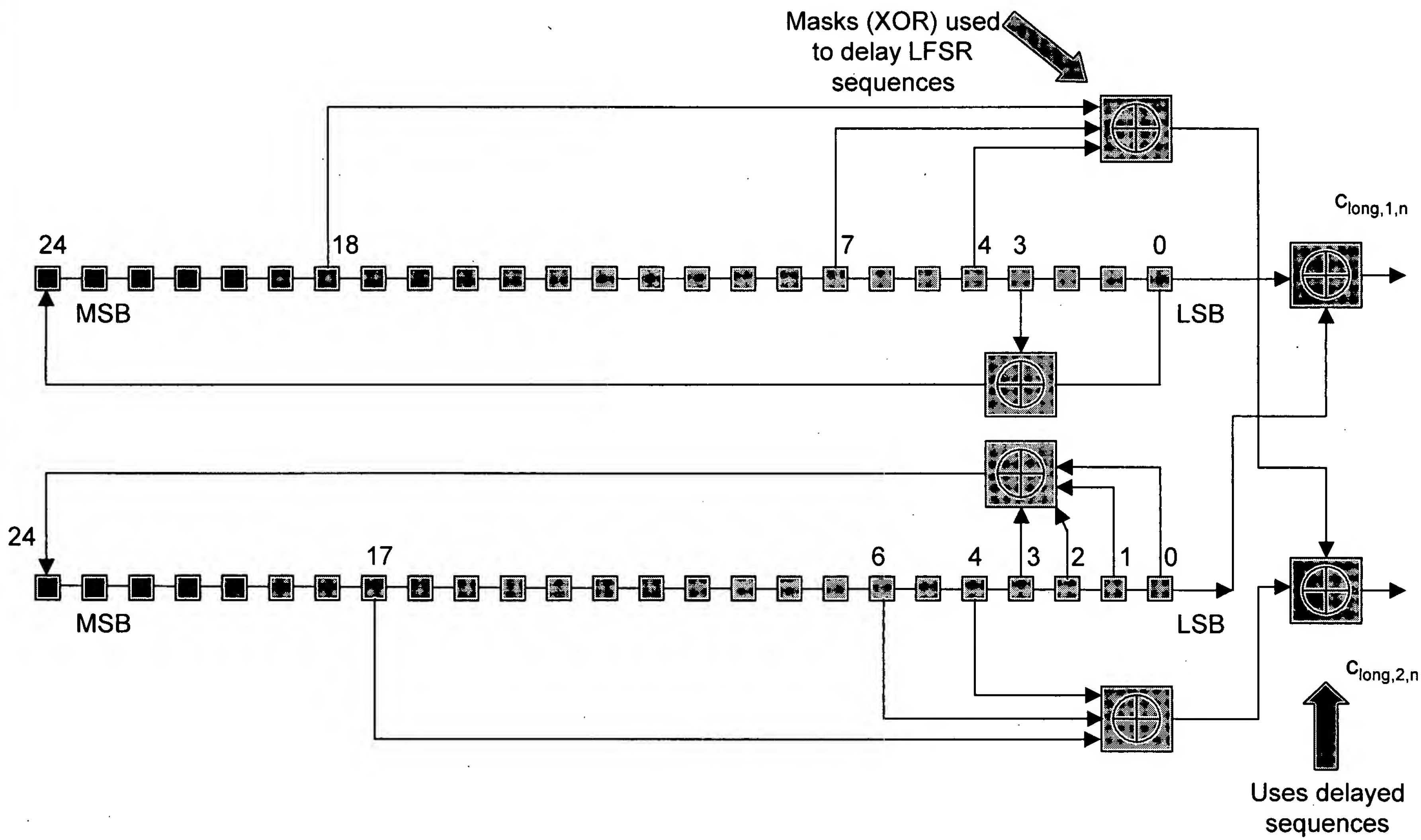
Antenna Sample Buffer Resource Requirements

- Implementation of:
  - 32 Users @ 125 MHz
  - 48 Users @ 187.5 MHz
  - 64 Users @ 250 MHz
  - ◆ 22 DPUs
  - ◆ 26 LSMs

UMTS Gold Code Generator Requirements & Assumptions

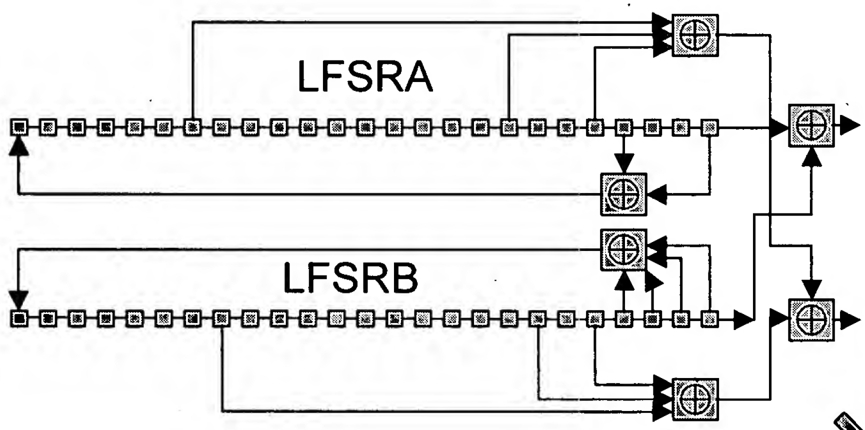
- Requirements
  - Needs to generate 16 bits per clock (8I, 8Q)
  - 32 Users at 125 MHz (The generator supports 64 users @125 MHz)
  - Easily scale to 64 users @ 250 MHz with single engine
  - 32-64 user expansion is only a function of memory
- Assumptions
  - The same Gold Code output may be shared by the different fingers of a single channel if the fingers are aligned
  - Each Gold Code is unique per user

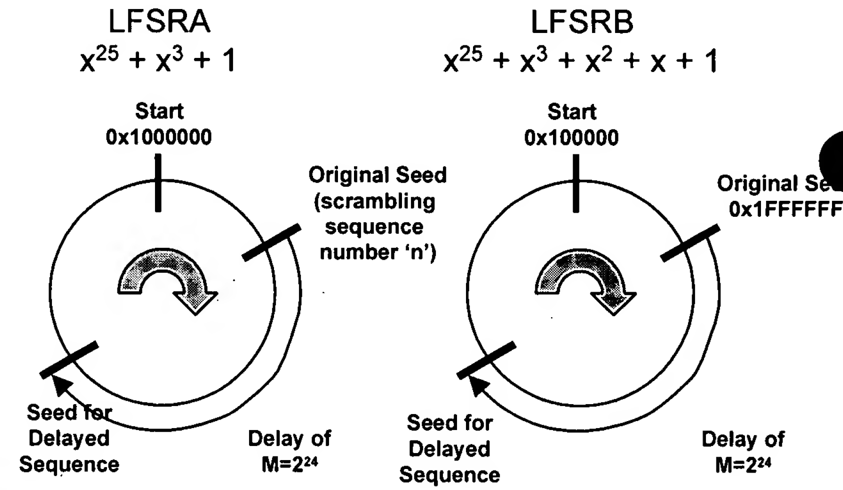
UMTS Gold Code Generator as defined by 3G TS 25.213



CHAMELEON SYSTEMS, INC.

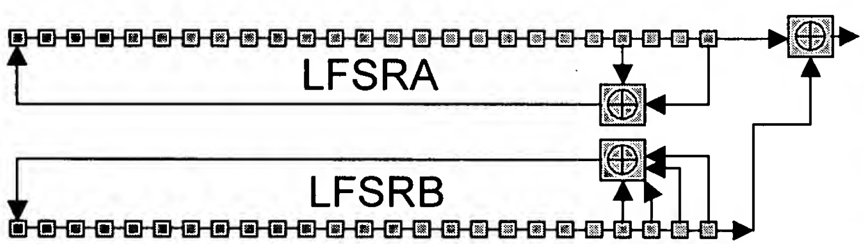
Exploit Code Properties to Eliminate Masks



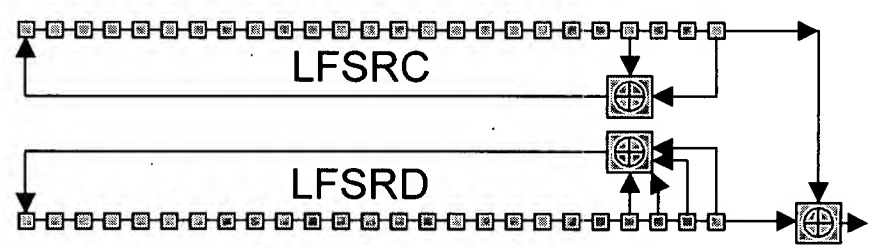


Implementation Unchanged

Use second pair of seeds to eliminate masks
(avoid XOR implementation)



LFSR Pair to Calculate $c_{long,1,n}$

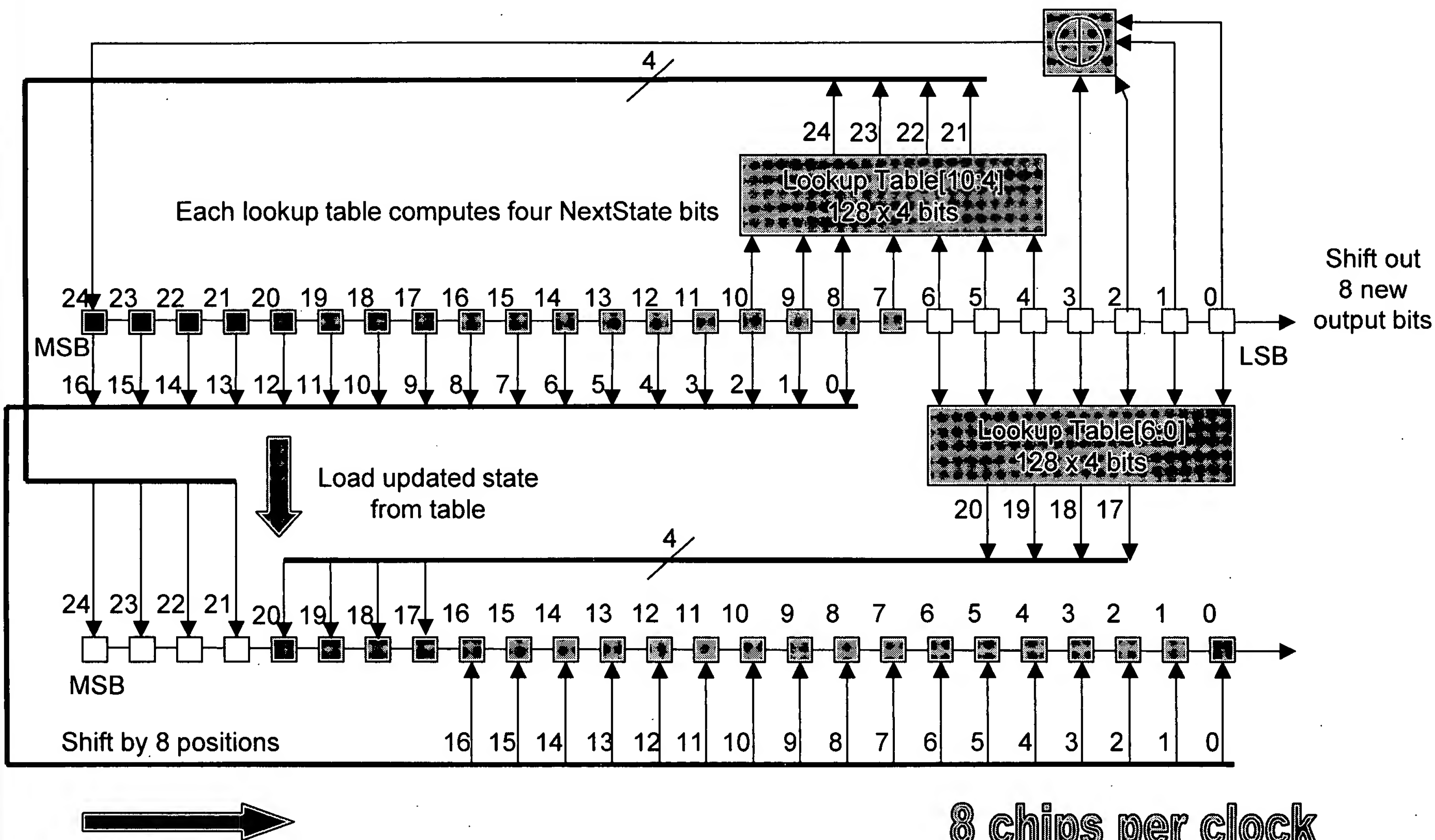


LFSR Pair to Calculate $c_{long,2,n}$

Linear Feedback Shift Register (LFSR) Initial Values

- Each LFSR is reset to its initial value at the beginning of the frame
- The seed for LFSRA is assigned by the Network Controller
- The seed for LFSRB is 0x1FFFFFF for all users
- The seed for LFSRC is the contents of LFSRA's seed shifted by 16,777,232 cycles, and is computed by the ARC at the beginning of the call
- The seed for LFSRD is 0x1FFFFFF shifted by 16,777,232 cycles for all users and is static
- The seed values for all LFSRs are stored in LSMs

Lookup Table Determines the Next 8 Bits In a Single Cycle



8 chips per clock

Gold Code Generator Output Computations

$C_{long1,n}$ = LSFRA[7:0] XOR LSFRB[7:0]
Let us define LFSRC'[i] = LSFRC[$2\lfloor i/2 \rfloor$]

$C_{long,n}(i) = C_{long1,n}(i)(1 + j(-1)^i(c_{long2},n(2\lfloor i/2 \rfloor)$ (from 3G TS25.213)

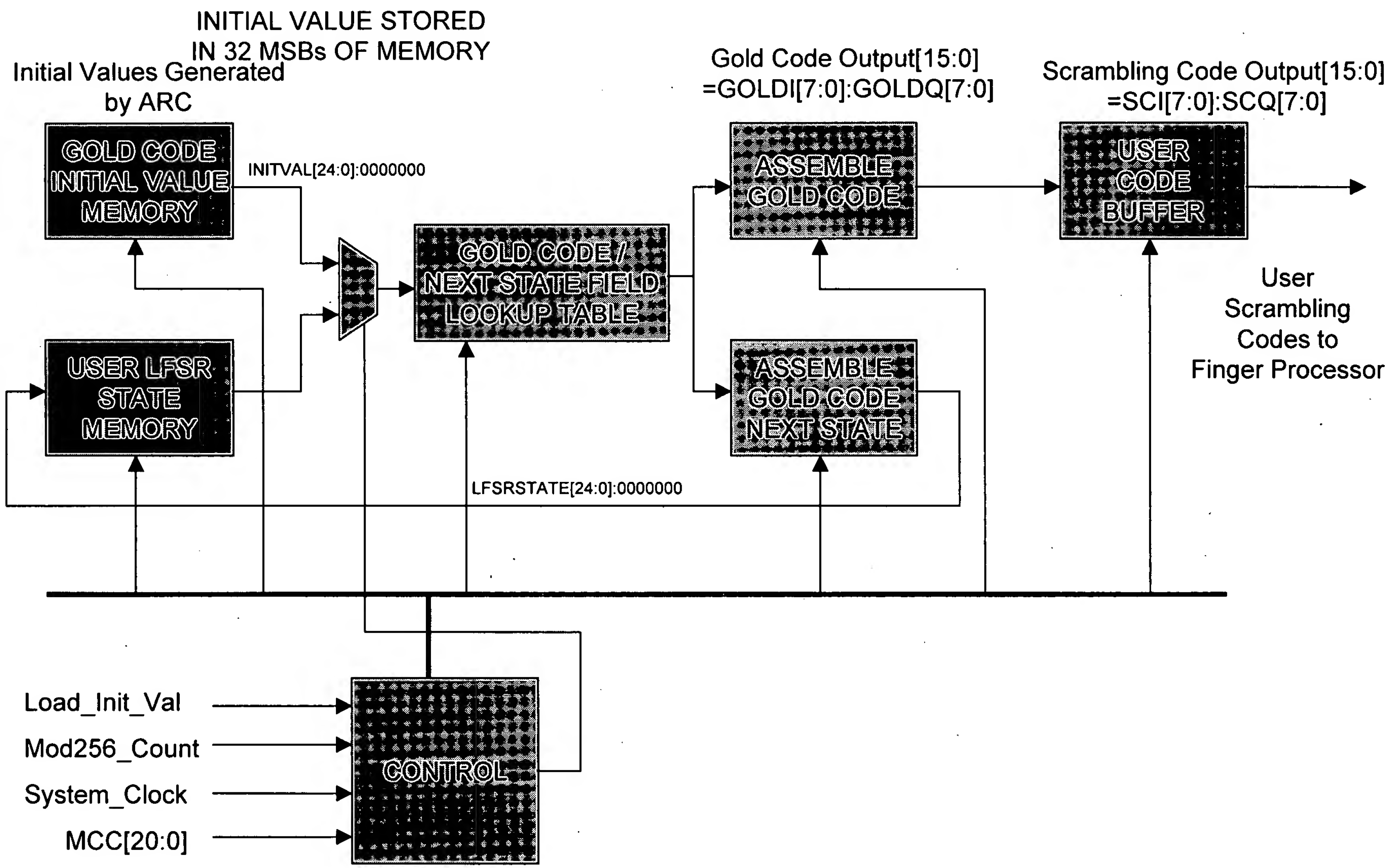
Multiplying bits by +1/-1 is the same as XOR for 0s and 1s.
XORing by 0xAA can be used in place of the $(-1)^i$ term.
In binary representation, the Scrambling Code $C_{long,n}$ becomes:

$C_{long,n}$[7:0] = $C_{long1,n}$[7:0](1 +j(0xAA) XOR $C_{long2,n}$[7:0])
$C_{long,n}$[7:0] = LFSRA[7:0] XOR LFSRB[7:0]
+j(LFSRA[7:0] XOR LFSRB[7:0] XOR 0xAA XOR LFSRC'[7:0] XOR LFSRD'[7:0]
$C_{long,n}$[7:0] = SCI[7:0] + jSCQ[7:0]

Let us define LFSRD"[7:0] = 0xAA XOR LFSRD'[7:0], then:
$C_{long,n}$[7:0] = (LFSRA[7:0] XOR LFSRB[7:0])
+ j(LFSRA[7:0] XOR LFSRB[7:0] XOR LFSRC'[7:0] XOR LFSRD"[7:0])

We use a lookup table to compute LFSRC'[7:0] and LFSRD"[7:0])

Gold Code Generator Functional Block Diagram



Gold Code Generator Memory Layout

Gold Code Initial Value Memory

Address	Contents
0xFC	User 63 LFSRD
0xF8	User 63 LFSRC
0xF4	User 63 LFSRB
0xF0	User 63 LFSRA
0xEC	User 62 LFSRD
0xE8	User 62 LFSRC
0xE4	User 62 LFSRB
0xE0	User 62 LFSRA
	⋮
0x1C	User 1 LFSRD
0x18	User 1 LFSRC
0x14	User 1 LFSRB
0x10	User 1 LFSRA
0x0C	User 0 LFSRD
0x08	User 0 LFSRC
0x04	User 0 LFSRB
0x00	User 0 LFSRA

User LFSR State Memory

Address	Contents
0xFC	User 63 LFSRD
0xF8	User 63 LFSRC
0xF4	User 63 LFSRB
0xF0	User 63 LFSRA
0xEC	User 62 LFSRD
0xE8	User 62 LFSRC
0xE4	User 62 LFSRB
0xE0	User 62 LFSRA
	⋮
0x1C	User 1 LFSRD
0x18	User 1 LFSRC
0x14	User 1 LFSRB
0x10	User 1 LFSRA
0x0C	User 0 LFSRD
0x08	User 0 LFSRC
0x04	User 0 LFSRB
0x00	User 0 LFSRA

User Code Buffer Memory

Address	Contents
0x1FE	User 63 PONG
0x1FC	User 62 PONG
0x1FA	User 61 PONG
	⋮
0x104	User 2 PONG
0x102	User 1 PONG
0x100	User 0 PONG
0xFE	User 63 PING
0x0C	User 62 PING
	⋮
0x04	User 2 PING
0x02	User 1 PING
0x00	User 0 PING

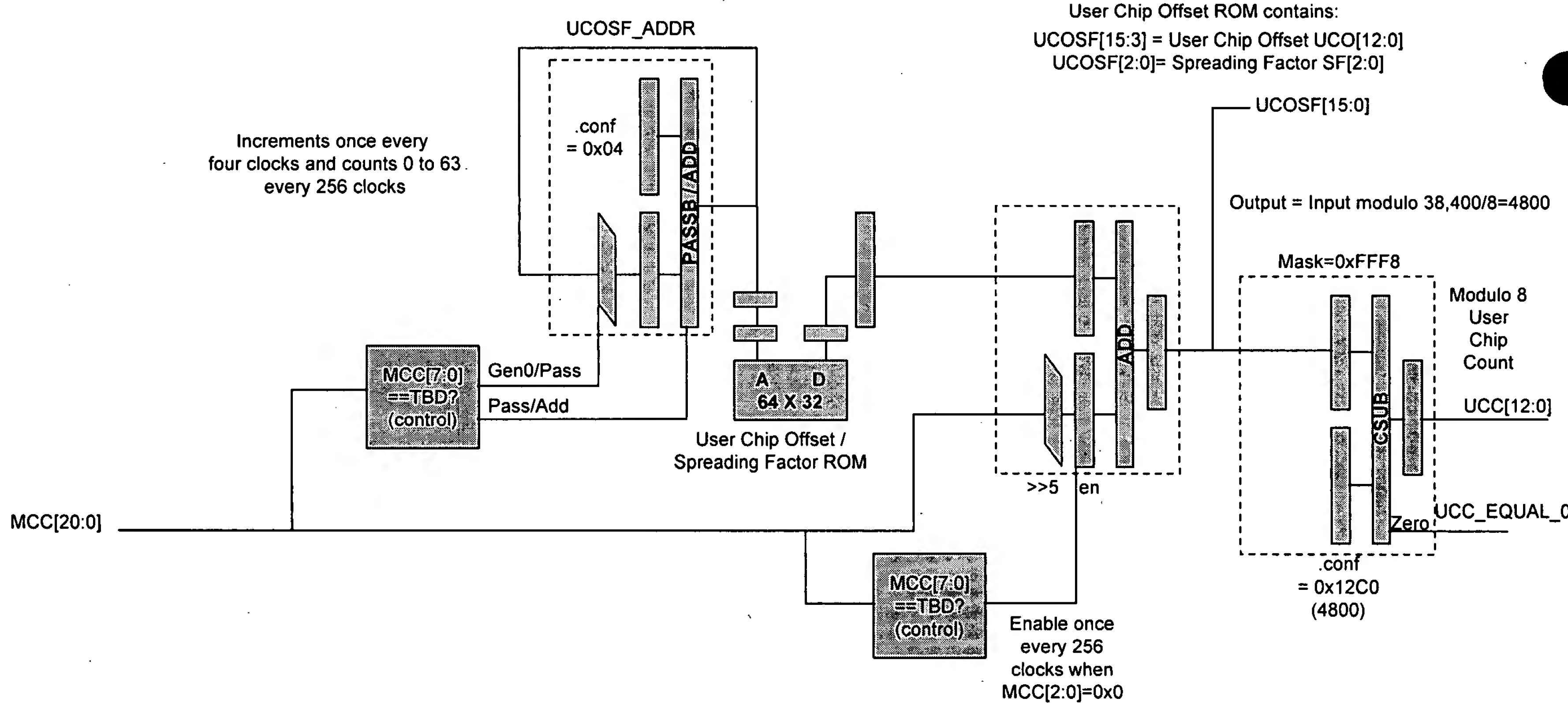
Gold Code Generator Lookup[6:0] Definitions

At Address 4n+0: OUT[7:0] = Next StateA[3:0]:PASSA[3:0] OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0] OUT[3] = IN[3] OUT[2] = IN[2] OUT[1] = IN[1] OUT[0] = IN[0]	At Address 4n+2: OUT[7:0] = Next StateC[3:0]:LFSRC'[3:0] OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0] OUT[3] = IN[2] OUT[2] = IN[2] OUT[1] = IN[0] OUT[0] = IN[0]
At Address 4n+1: OUT[7:0] = Next StateB[3:0]:PASSB[3:0] OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0] OUT[3] = IN[3] OUT[2] = IN[2] OUT[1] = IN[1] OUT[0] = IN[0]	At Address 4n+3: OUT[7:0] = Next StateD[3:0]:LFSRD"[3:0] OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0] OUT[3] = /IN[2] OUT[2] = IN[2] OUT[1] = /IN[0] OUT[0] = IN[0]

Gold Code Generator Lookup[10:4] Definitions

At Address 4n+0: OUT[7:0] = IN[7:4]:Next StateA[7:4] OUT[7] = IN[3] OUT[6] = IN[2] OUT[5] = IN[1] OUT[4] = IN[0] OUT[3] = IN[6] XOR IN[3] OUT[2] = IN[5] XOR IN[2] OUT[1] = IN[4] XOR IN[1] OUT[0] = IN[3] XOR IN[0]	At Address 4n+2: OUT[7:0] = IN'[7:4]:Next StateC[7:4] OUT[3] = IN[2] OUT[2] = IN[2] OUT[1] = IN[0] OUT[0] = IN[0] OUT[7] = IN[6] XOR IN[3] OUT[6] = IN[5] XOR IN[2] OUT[5] = IN[4] XOR IN[1] OUT[4] = IN[3] XOR IN[0]
At Address 4n+1: OUT[7:0] = IN[7:4]:Next StateB[7:4] OUT[7] = IN[3] OUT[6] = IN[2] OUT[5] = IN[1] OUT[4] = IN[0] OUT[3] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[2] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[1] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[0] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0]	At Address 4n+3: OUT[7:0] = IN"[7:4]:Next StateD[7:4] OUT[3] = /IN[2] OUT[2] = IN[2] OUT[1] = /IN[0] OUT[0] = IN[0] OUT[7] = IN[6] XOR IN[5] XOR IN[4] XOR IN[3] OUT[6] = IN[5] XOR IN[4] XOR IN[3] XOR IN[2] OUT[5] = IN[4] XOR IN[3] XOR IN[2] XOR IN[1] OUT[4] = IN[3] XOR IN[2] XOR IN[1] XOR IN[0]

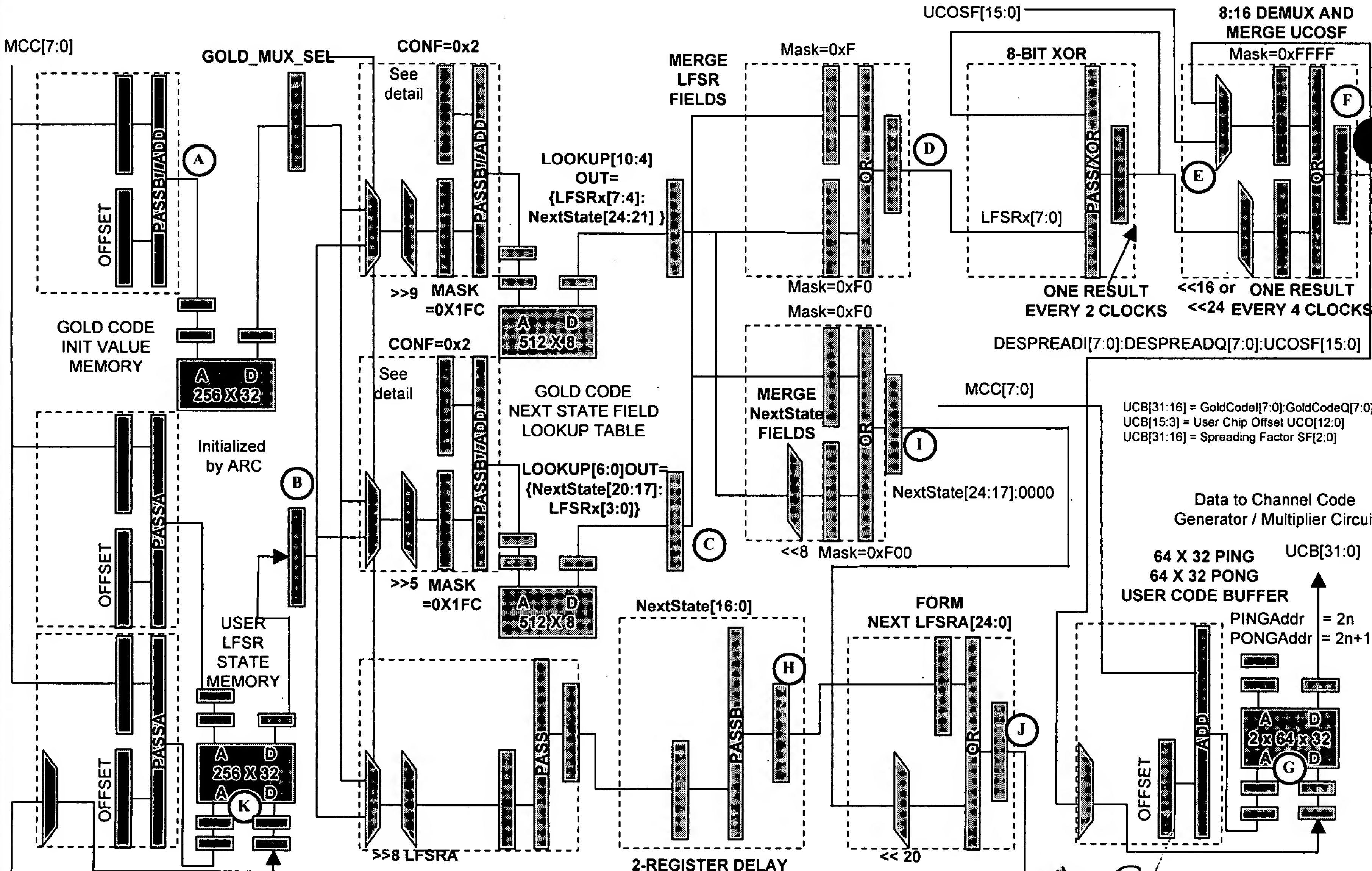
Gold Code Control Implementation



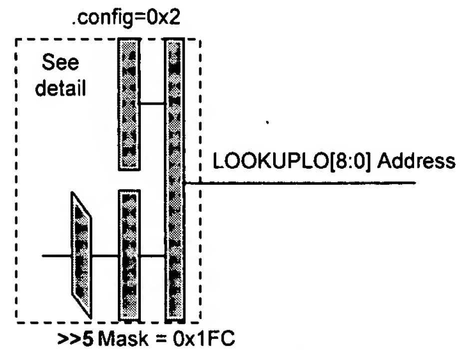
Add User Chip Offset
to Master Chip Counter

If MCC[7:0]= TBD ;Address=0
ALUB = shifterconstant = 0
ALU = PASSB
else ;Address = Address + 4
ALUB = BREG
ALU = ADD

Gold Code Generator Implementation



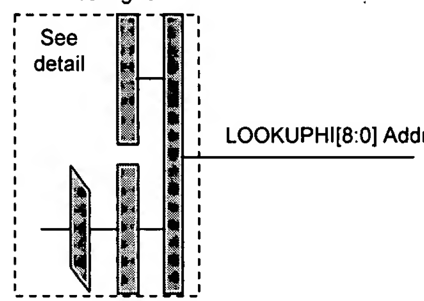
Gold Code Generator Implementation Lookup[6:0] Address Generation DPU



Control needs to be able to select one of four lookup
tables in the RAM.
Each lookup table is selected by two control inputs Select[1:0]:
Select[1:0] resides in the bottom two bits of the address
If Select[1:0]=0x0 then
 ALU=PASSB
 Address[1:0]=0x0
 Output[7:4]= NextStateA[20:17], Output[3:0]=LFSRA[3:0]
If Select[1:0]=0x1 then
 ALU=INC (B+1)
 Address[1:0]=0x1
 Output[7:4]= NextStateB[20:17], Output[3:0]=LFSRB[3:0]
If Select[1:0]=0x2 then
 ALU=ADD (A + B)
 Address[1:0]=0x2
 Output[7:4]= NextStateC[20:17], Output[3:0]=LFSRC'[3:0]
If Select[1:0]=0x3 then
 ALU=ADDCNT (A + B + 1, force Cin by control)
 Address[1:0]=0x3
 Output[7:4]= NextStateD[20:17], Output[3:0]=LFSRD''[3:0]

Select[1:0] is a function of the Master Chip Counter bits MCC[1:0]

Gold Code Generator Implementation Lookup[10:4] Address Generation DPU



Control needs to be able to select one of four lookup tables in the RAM.
Each lookup table is selected by two control inputs Select[1:0]:
Select[1:0] resides in the bottom two bits of the address
If Select[1:0]=0x0 then
  ALU=PASSB
  Address[1:0]=0x0
  Output[7:4]= LFSRA[7:4], Output[3:0]= NextStateA[24:21]
If Select[1:0]=0x1 then
  ALU=INC (B+1)
  Address[1:0]=0x1
  Output[7:4]= LFSRB[7:4], Output[3:0]= NextStateB[24:21]
If Select[1:0]=0x2 then
  ALU=ADD (A + B)
  Address[1:0]=0x2
  Output[7:4]= LFSRC'[7:4], Output[3:0]= NextStateC[24:21]
If Select[1:0]=0x3 then
  ALU=ADDCNT (A + B + 1, force C_{in} by control)
  Address[1:0]=0x3
  Output[7:4]= LFSRD''[7:4], Output[3:0]= NextStateD[24:21]

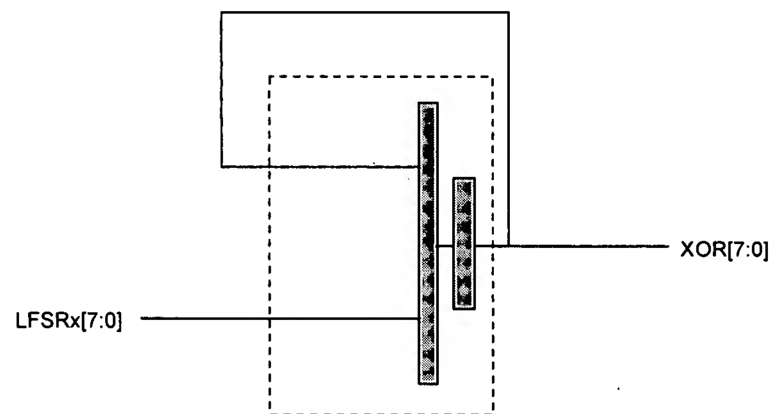
Select[1:0] is a function of the Master Chip Counter bits MCC[1:0]

CHAMELEON SYSTEMS, INC.

Gold Code Generator Implementation 8-Bit XOR DPU

D	LFSR Input	A	B	C	D	A	B	C	D	A	B
	XOR DPU ALU Instruction	PASSB	XOR	XOR	XOR	PASSB	XOR	XOR	XOR	PASSB	XOR
E	XOR Output Register	A^B^C^D	A	A^B	A^B^C	A^B^C^D	A	A^B	A^B^C	A^B^C^D	A

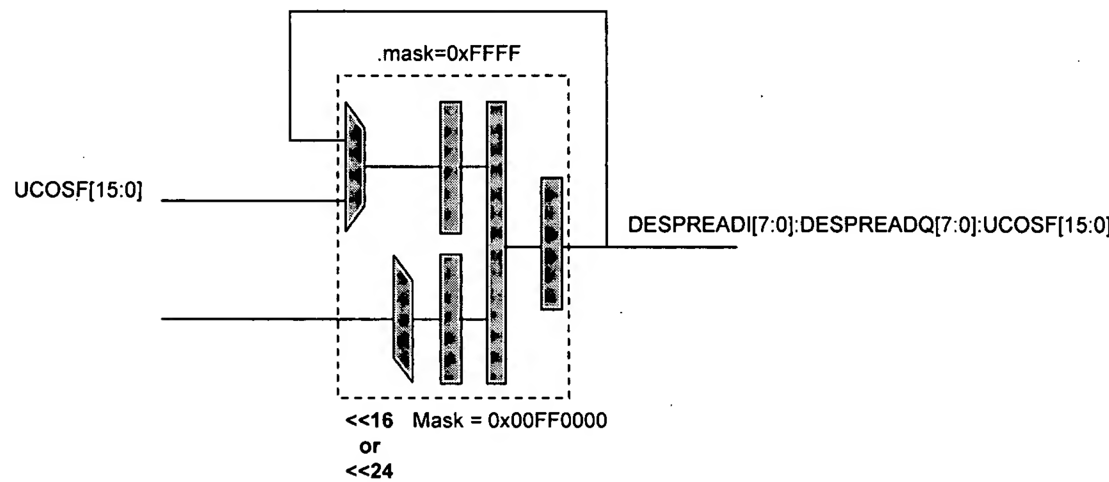
Where:
A = LFSRA[7:0]
B = LFSRB[7:0]
C = LFSRC'[7:0]
D = LFSRD'[7:0]
^ = XOR



This DPU is used to perform an XOR operation

Control needs to generate two states for the DPU:
If PASS
;Pass the input to the output register
ALUB=LFSRx[7:0] (from previous stage)
ALU = PASSB
OUTREGEN = 1
Else
;XOR the input with the output register
ALUA = OUTREG
ALUB=LFSRx[7:0] (from previous stage)
ALU = XOR
OUTREGEN = 1

Gold Code Generator Implementation 8:16 Demultiplex and Merge UCOSF DPU



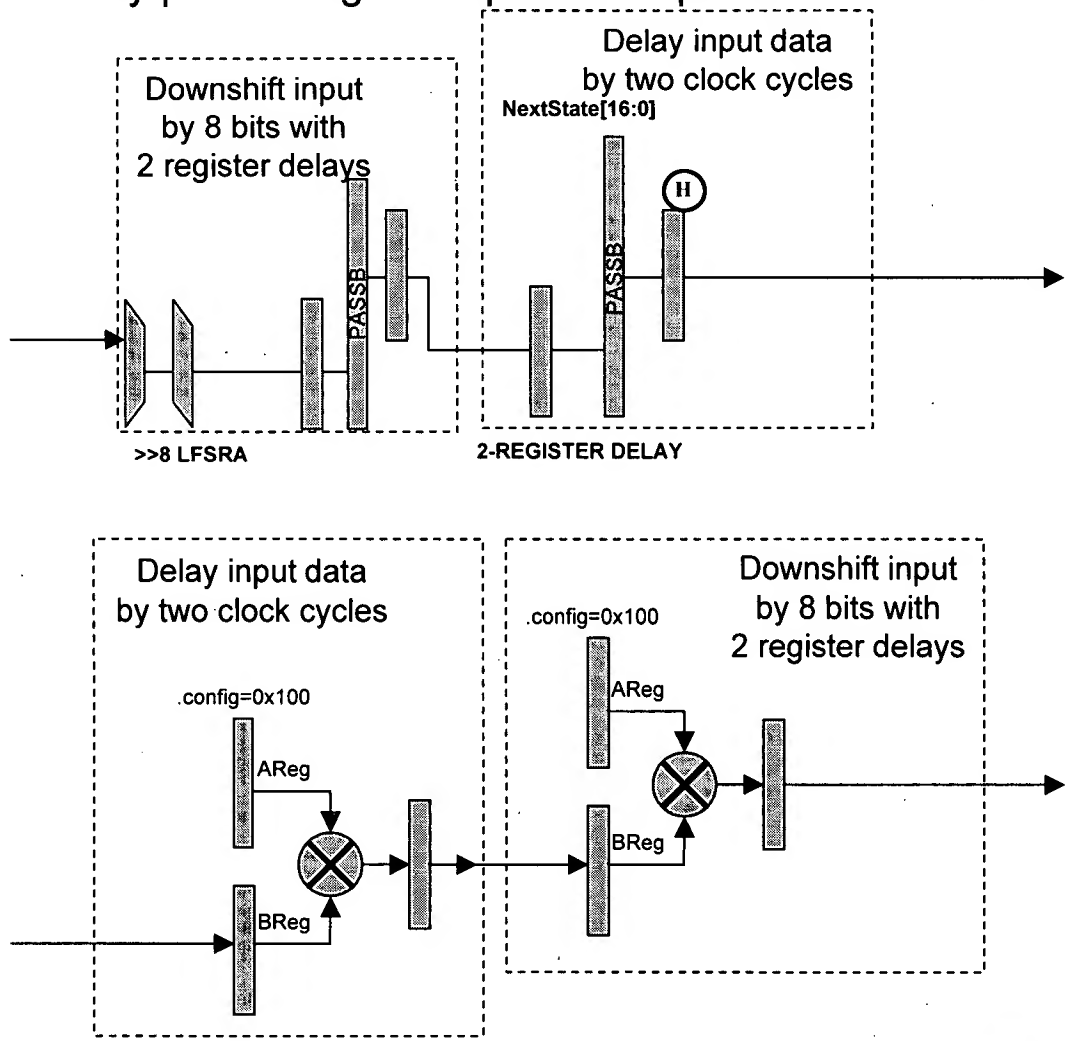
This DPU is used to perform a 8:16 demultiplexing operation on the LSFR data as well as merge the UCOSF[15:0] field into the data stream before it is written into the User Code Buffer RAM.

Control needs to generate three states for the DPU:

```
If MERGE_I
   ;Place UCOSF[15:0] in bits [15:0]
   ;and place DESPREADI[7:0] in bits [31:24]
   ALUA=UCOSF[15:0] && (Mask==0xFFFF)
   ALUB=(DESPREADI[7:0] from previous stage)<<24
   ALU = OR
   OUTREGEN = 1
If MERGE_Q
   ;Merge DESPREADQ[7:0] into bit positions [23:16]
   ALUA = OUTREG
   ALUB =( (DESPREADQ[7:0] from previous stage)<<16) && Mask==0x00FF0000
   ALU = OR
   OUTREGEN = 1
Else
   OUTREGEN = 0
```

Gold Code Generator Implementation of Downshift Pipeline Delay with Multipliers

Two DPUs may be saved by performing the equivalent operations with two multipliers



00220T"16586560

Gold Code Generator Timing

A	Mod256_Count	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C	U2D	U3A	U3B	U3C	U3D	U4A	U4B	U4C	U4D
B	User LFSR State Output	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C	U2D	U3A	U3B	U3C	U3D
C	Gold Code Lookup Output	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C	U2D
D	Merge LFSRx[7:0] Fields	U61D	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C
E	XOR LFSRx[7:0] Fields		U61Q		U62I		U62Q		U63I		U63Q		U0I		U0Q		U1I		U1Q		U2I
F	Merge Scrambling Codes			U61IQ				U62IQ				U63IQ				U0IQ				U1IQ	
G	User to Finger Buffer Input	U60IQ				U61IQ				U62IQ				U63IQ				U0IQ			
H	Form NextState LFSRx[16:0]	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C	U2D
I	Merge NextState Fields	U61D	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B	U2C
J	Form NextState LFSRx[24:0]	U61C	U61D	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D	U2A	U2B
K	User LFSR State Memory Input	U61A	U61B	U61C	U61D	U62A	U62B	U62C	U62D	U63A	U63B	U63C	U63D	U0A	U0B	U0C	U0D	U1A	U1B	U1C	U1D

Gold Code Data Format:
Gold Code Output[15:0] =GOLDI[7:0]:GOLDQ[7:0]

00220T"h6986960

Gold Code Generator UCOSF Output Timing

	UCOSF[15:0]	U0	U0	U62	U62	U1	U1	U63	U63	U2	U2	U0	U0	U3	U3	U1	U1	U4	U4	U2	U2
	UCC[12:0], UCC_EQUAL_0	XX	U0	U0	XX	XX	U1	U1	XX	XX	U2	U2	XX	XX	U3	U3	XX	XX	U4	U4	XX
	REG_UCC_EQUAL_0	XX	XX	U0	U0	U0	U0	U1	U1	U1	U1	U2	U2	U2	U2	U3	U3	U3	U3	U4	U4
B	GOLD_MUX_SEL	U63	U63	U63	U0	U0	U0	U0	U1	U1	U1	U1	U2	U2	U2	U2	U3	U3	U3	U3	U4
E	Gold Code 8-BIT XOR	U61B		U62A		U62B		U63A		U63B		U0A		U0B		U1A		U1B		U2A	
E	UCOSF[15:0]			U62				U63				U0				U1				U2	
F	8:16 Demux Output		U61		U62 temp		U62		U63 temp		U63		U0 temp		U0		U1 temp		U1		U2 temp

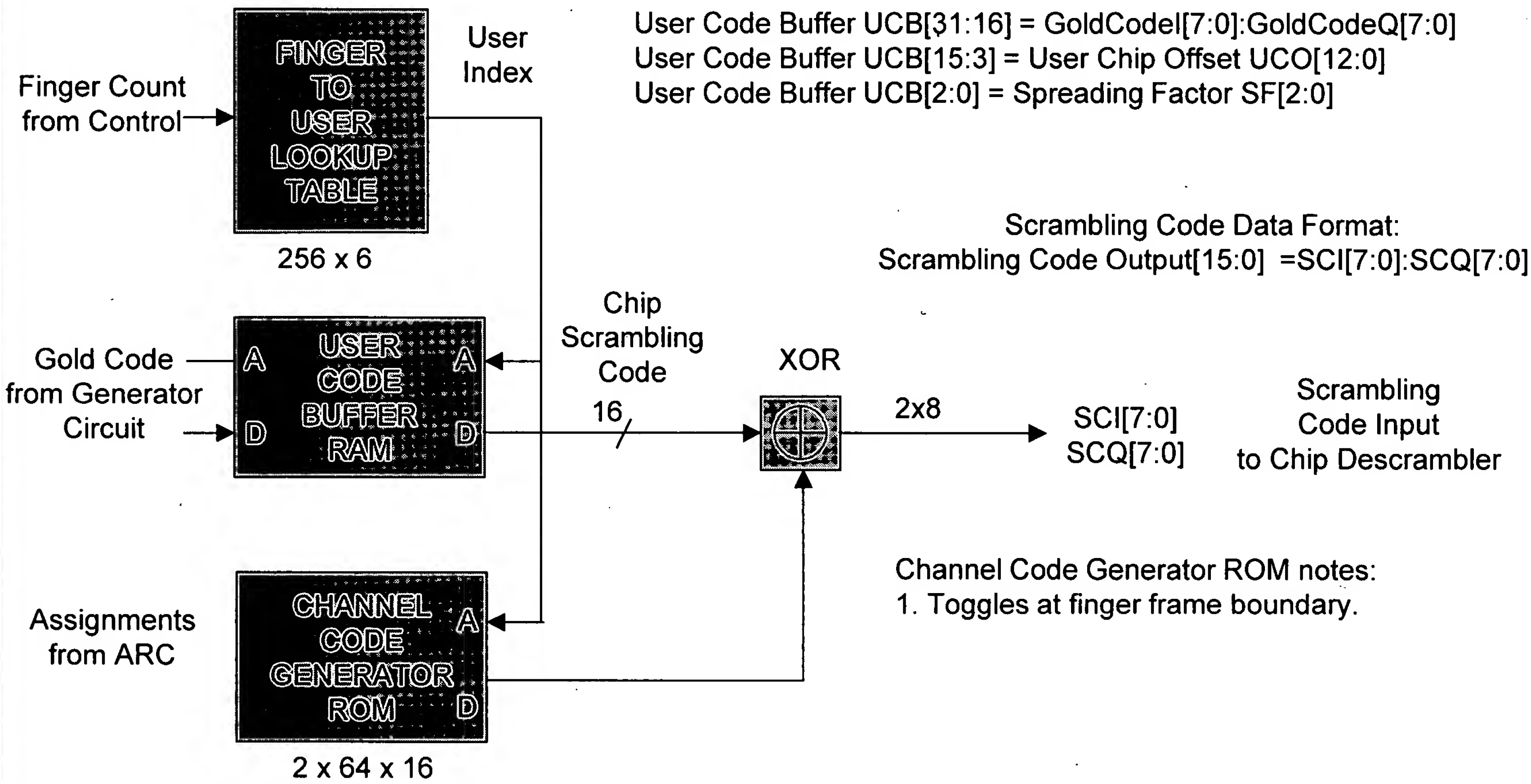
Gold Code Generator Resource Requirements

- 32 User Implementation @ 125 MHz
  - 14 DPUs
  - 8 LSMs
  - 2 Multipliers
- 64 User Implementation @ 125 MHz
  - 14 DPUs
  - 10 LSMs
  - 2 Multipliers
- 128 User Implementation @ 250 MHz
  - 16 DPUs
  - 12 LSMs
  - 2 Multipliers

Channel Code Generator / Multiplier Requirements and Assumptions

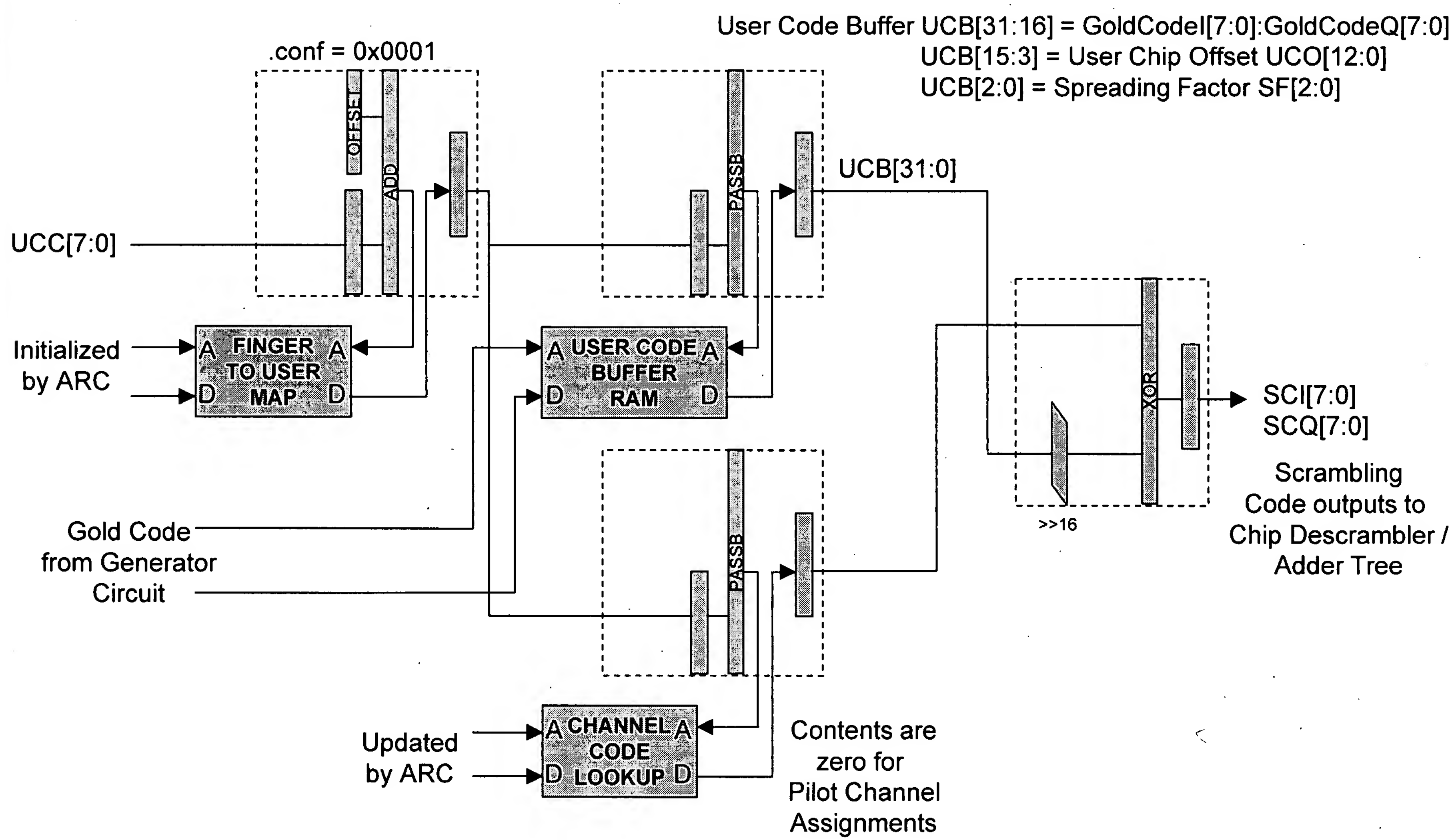
- Requirements
  - Provide a User to Finger Interface
  - Convert between <u>User-based</u> Gold Code and <u>Finger-based</u> Chip despreading
- Assumptions (from 3G TS25.213 Specification)
  - DPCCH $c_c = c_{ch,256,0} = 1, 1, 1, 1, \ldots$ (all ones)
  - Single DPDCH $c_{d,1} = c_{ch,SF,k}$ where k=SF/4
    - $c_{ch,4,1} = 1, 1, -1, -1, \ldots$
    - $c_{ch,8,2} = 1, 1, -1, -1, \ldots$
    - $c_{ch,16,4} = 1, 1, -1, -1, \ldots$
  - If more than one $DPDCH_n$ SF=4, $c_{d,n} = c_{ch,4,k}$
    - k=1 if $n \in \{1,2\}$
    - k=3 if $n \in \{3,4\}$
    - k=2 if $n \in \{5,6\}$

Channel Code Generator / Multiplier Functional Block Diagram



Channel Code Generator ROM notes:
1. Toggles at finger frame boundary.

Channel Code Generator / Multiplier Implementation



CHAMELEON SYSTEMS, INC.

Channel Code Generator / Multiplier Resource Requirements

- Implementation of:
  - 32 Users @ 125 MHz
  - 48 Users @ 187.5 MHz
  - 64 Users @ 250 MHz
  - 4 DPUs
  - 2 LSMs
- Note that the Gold Code Output Buffer LSM resources are counted in the Gold Code Generator circuit

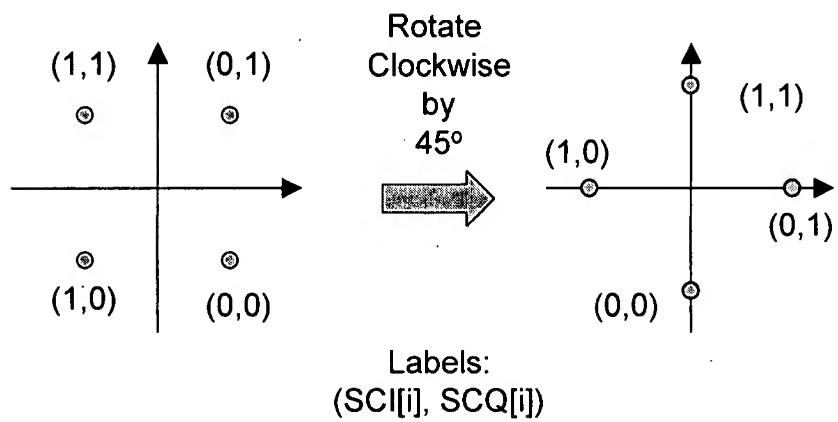
Chip Descrambler Requirements and Assumptions

- Requirements
  - Include the Adder Tree Input multiplexer
  - Descramble 8 chips per clock
  - Process 256 fingers @ 125 MHz
  - Process 512 fingers @ 250 MHz
- Assumptions
  - I must be able to read 8 consecutive samples (T_c apart) per clock from the Antenna Sample Buffer from any one antenna
  - An 8:1 Multiplexer is needed to align the input data to an even SF boundary point
  - All fingers with SF=4 will require 2 consecutive finger assignments so that two sums of four chips may be routed to the output of the Adder Tree in two clocks

Chip Descrambler Functional Description

- Read sixteen consecutive $T_c/2$ samples out of the Antenna Buffer corresponding to one finger, at the offset specified by the Path Searcher
- Mask out the unwanted half-chip samples
- Align the remaining eight samples (32 x 8 Barrel Shifter) with the Descrambling Code (Gold Code)
- Sign-extend the 8-bit data samples to 16 bits
- Multiply the eight aligned samples with the appropriate Despreading Codes

Chip Descrambler Functional Description

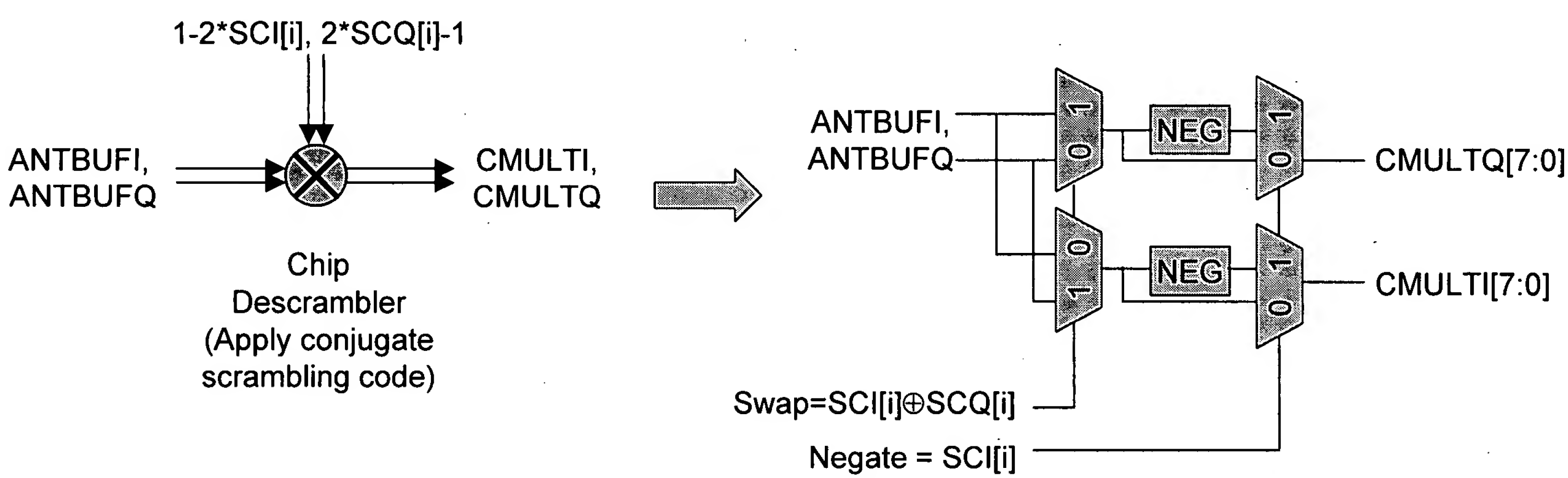


IN[31:16]=Q[15:0], IN[15:0]=I[15:0]

SCI	SCQ	OUT[31:16]	OUT[15:0]	SWAP/ CONJUGATE	NEGATE
0	0	-I	Q	Conjugate	
0	1	Q	I	Swap	
1	0	-Q	-I	Swap	Negate
1	1	I	-Q	Conjugate	Negate

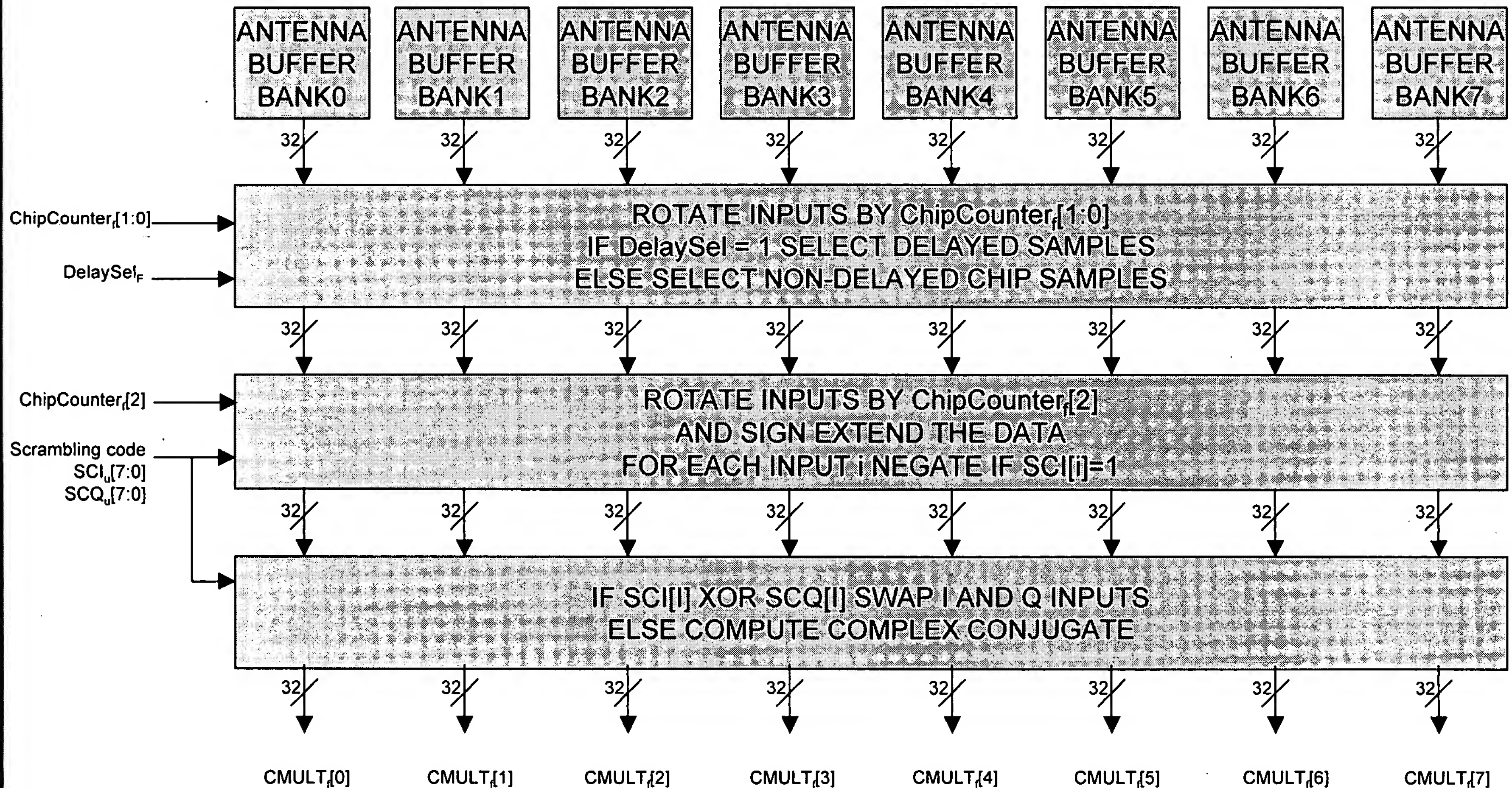
OUT[31:16]= CMULTI[15:0], OUT[15:0]=CMULTQ[15:0]

No multiplies or adds required

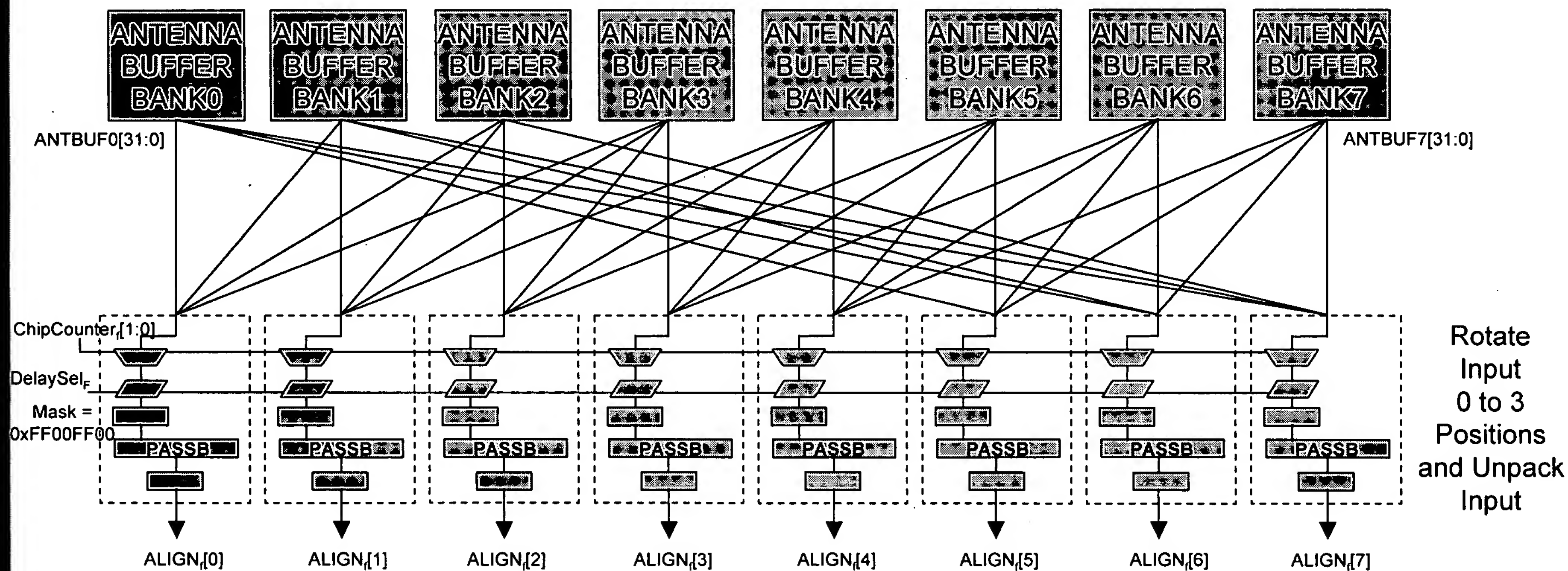


00220T' t6986960

Chip Descrambler Block Diagram



Chip Descrambler
Input Alignment Implementation

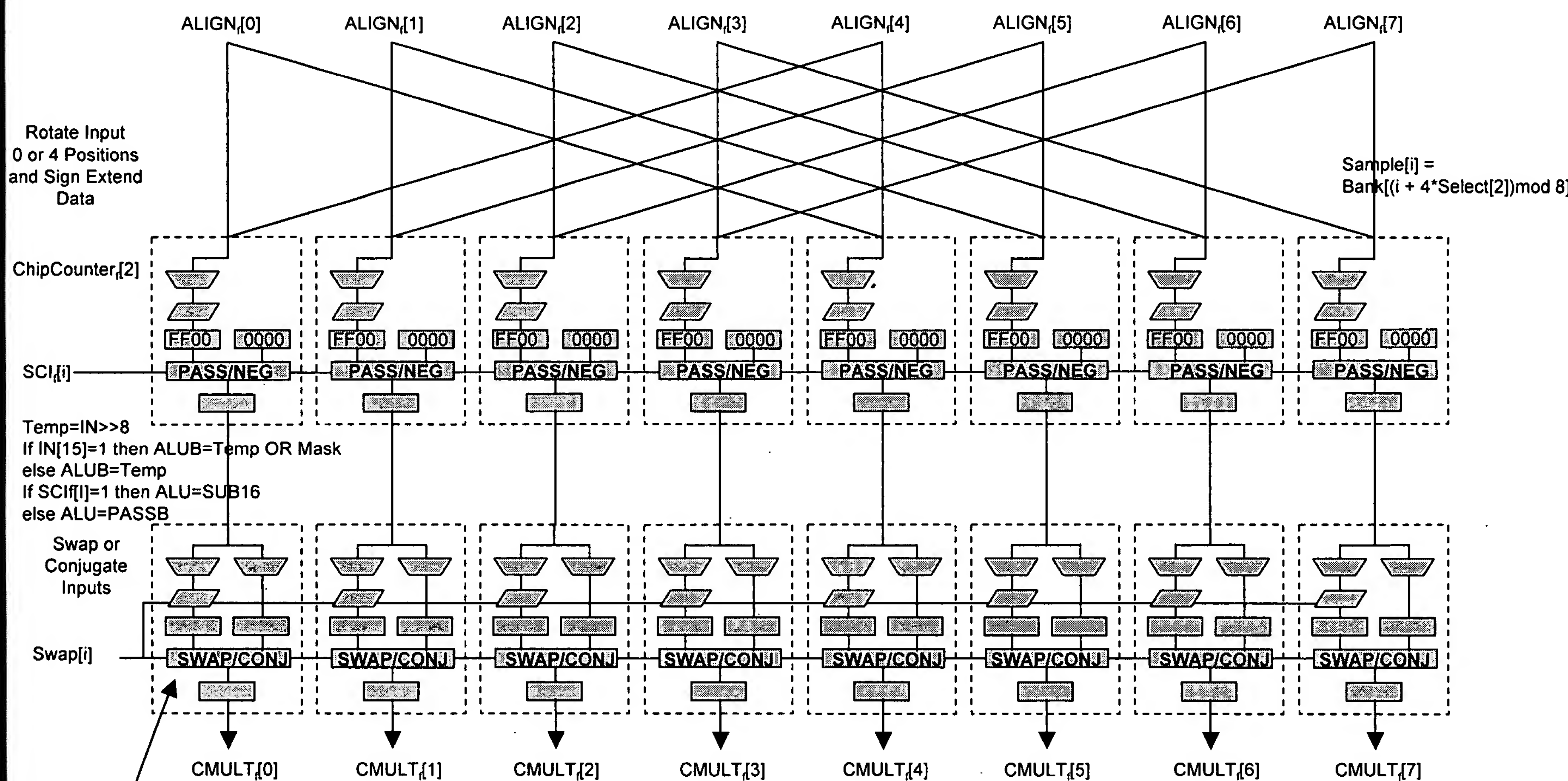


Outputs to Conjugate Unit

Shifter Control
for Each Cell:
DelaySel$_F$=0: <<0
DelaySel$_F$ =1: <<8

Input Multiplexer Control:
ALIGN$_f$[i] = Bank[(i + Select[1:0])mod 8]

Chip Descrambler
Chip Multiplier Implementation



Swap Control for Each Cell:
If Swap[i] = 0
then ALUA[31:0]=IN[31:0],ALUB[31:0] =(IN[31:0]<<1 AND 0XFFFF)
ALU=SUB16
else ALUA[31:0]=IN[31:0],ALUB[15:0]=IN[31:16],ALUB[31:16]=IN[15:0]
ALU=PASSB

Output to Swap/Conjugate Unit

CHAMELEON SYSTEMS, INC.

Chip Descrambler Data Organization

Since the DPUs can only perform a SWAP or a Complex Conjugate, the I and Q inputs are assumed to be preswapped at the input to the Chip Descrambler circuit.

	SAMPLE[31:24]	SAMPLE[23:16]	SAMPLE[15:8]	SAMPLE[7:0]
SAMPLE BUFFER OUTPUTS: ANTBUF[31:0]	DQ_n[7:0]	DelayedDQ_n[7:0]	DI_n[7:0]	DelayedDI_n[7:0]
INPUT ALIGNMENT OUTPUTS: $ALIGN_F$[31:0]	DQ[7:0]	0x00	DI[7:0]	0x00
CHIP MULTIPLIER OUTPUTS: $CMULT_F$[31:0]	DI_n[15:0]	DI_n[7:0]	DQ_n[15:8]	DQ_n[7:0]
ADDER TREE OUTPUTS: ADD_F[31:0]	DI_n[15:0]	DI_n[7:0]	DQ_n[15:8]	DQ_n[7:0]

Where n is the sample n and n+ 1/2 is the next half-chip sample out of the Antenna Sample Buffer

Chip Descrambler Control Implementation

Control Input:
Scrambling Code Input $SCI_m[7:0]$, $SCQ_m[7:0]$
for User m

Data Inputs:
Eight Samples $DI_i[7:0]$
for k = 0 to 7

$SCI_n[i]$	$SCQ_n[i]$	$OUTI_m[i]$	$OutQ_m[i]$
0	0	$DQ_k[7:0]$	$-DI_k[7:0]$
0	1	$DI_k[7:0]$	$DQ_k[7:0]$
1	0	$-DI_k[7:0]$	$-DQ_k[7:0]$
1	1	$-DQ_k[7:0]$	$DI_k[7:0]$

Chip Descrambler Resource Requirements

- Implementation of:

  32 Users @ 125 MHz

  48 Users @ 187.5 MHz

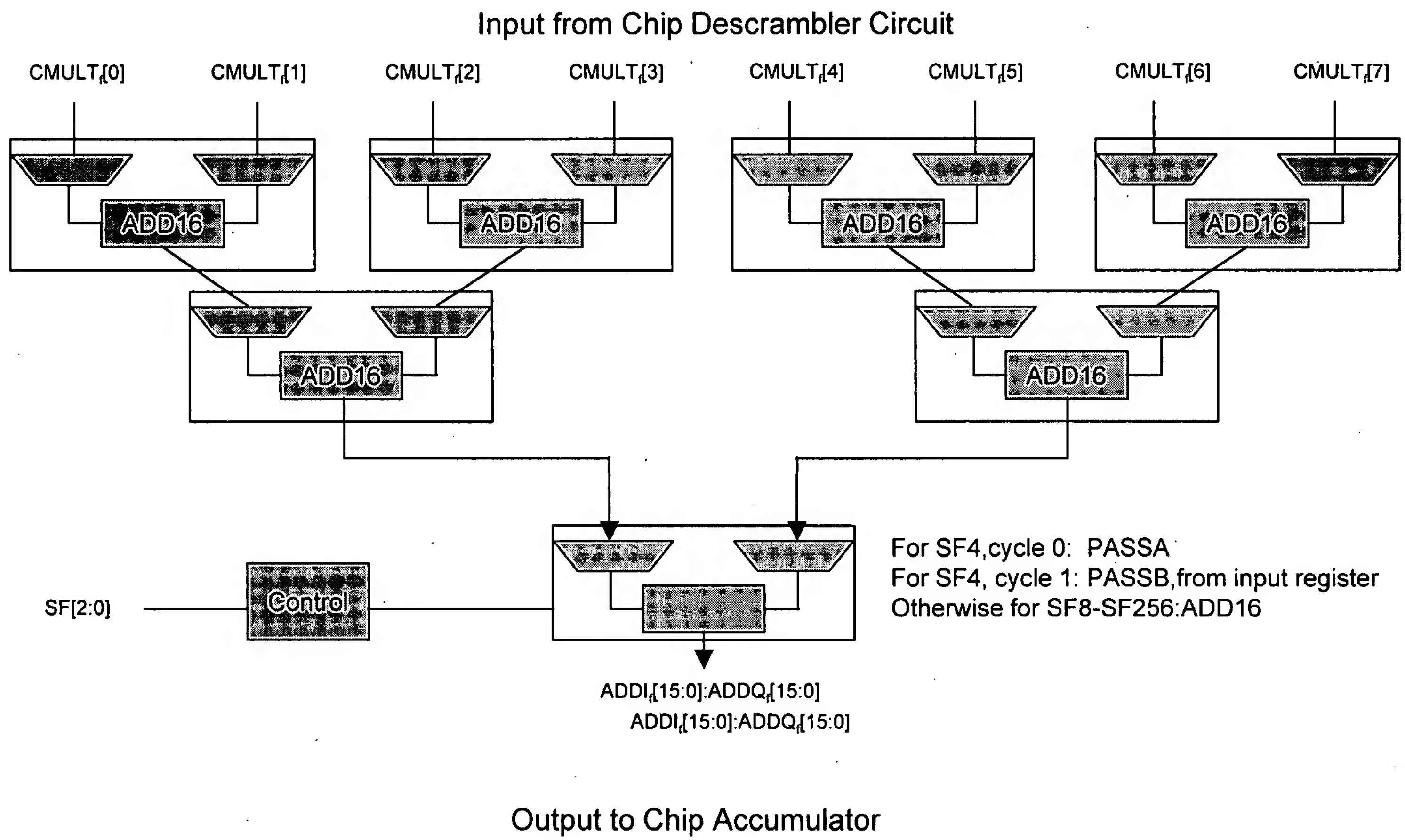
  64 Users @ 250 MHz

  - 24 DPUs
  - 0 LSMs

Chip Adder Tree Requirements and Assumptions

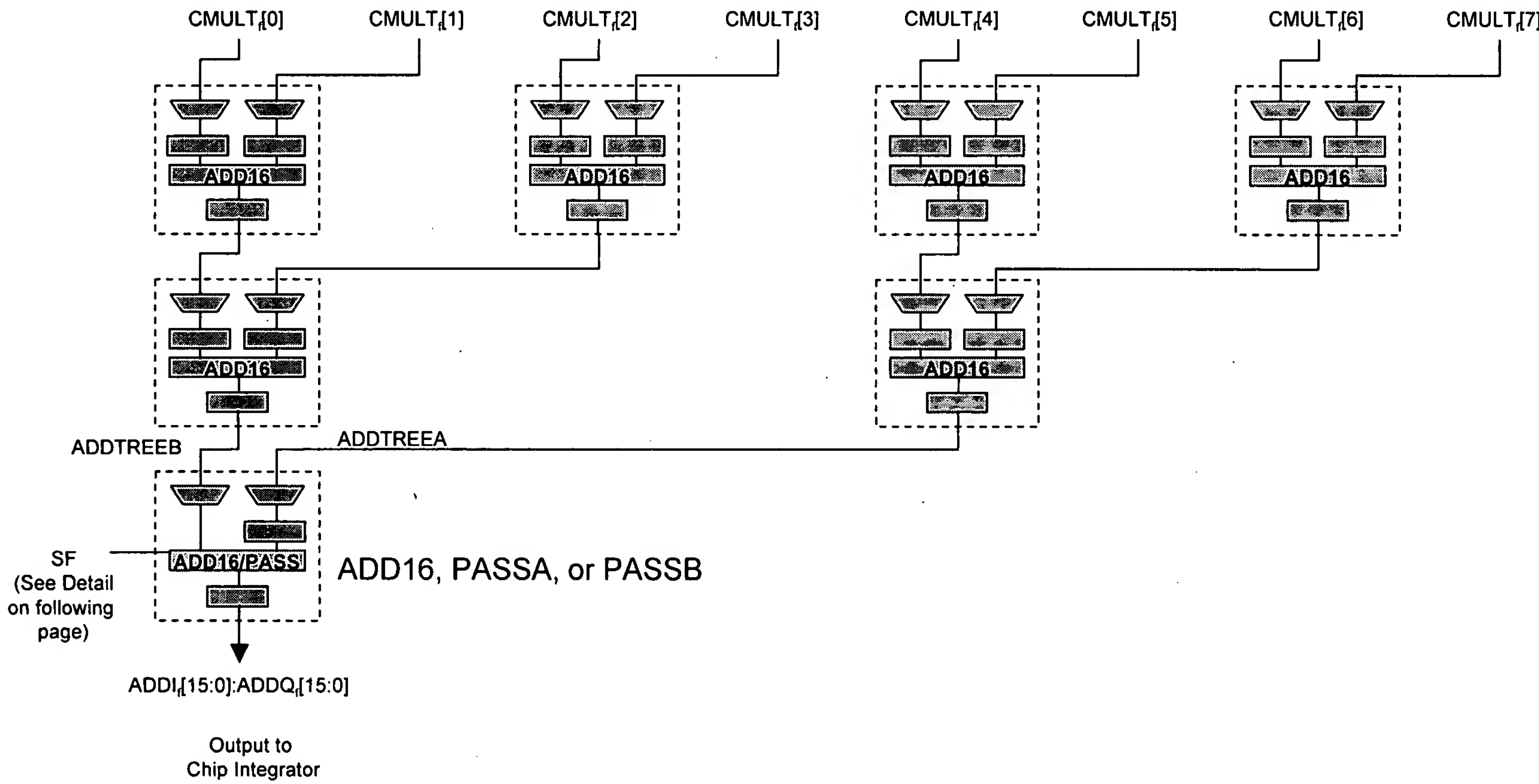
- Requirements
  - Add 8 chips per clock
  - Process 256 fingers @ 125 MHz
  - Process 512 fingers @ 250 MHz
- Assumptions
  - Inputs have been aligned on an even 8-chip boundary by the shifter in the Chip Descrambler circuit
  - All fingers with SF=4 will require 2 consecutive finger assignments so that two sums of four chips may be routed to the output of the Adder Tree in two clocks
  - Finger pairs allocated for SF=4 require that the first finger is assigned SF=4 and the second finger is assigned SF=0

Chip Adder Tree Functional Block Diagram



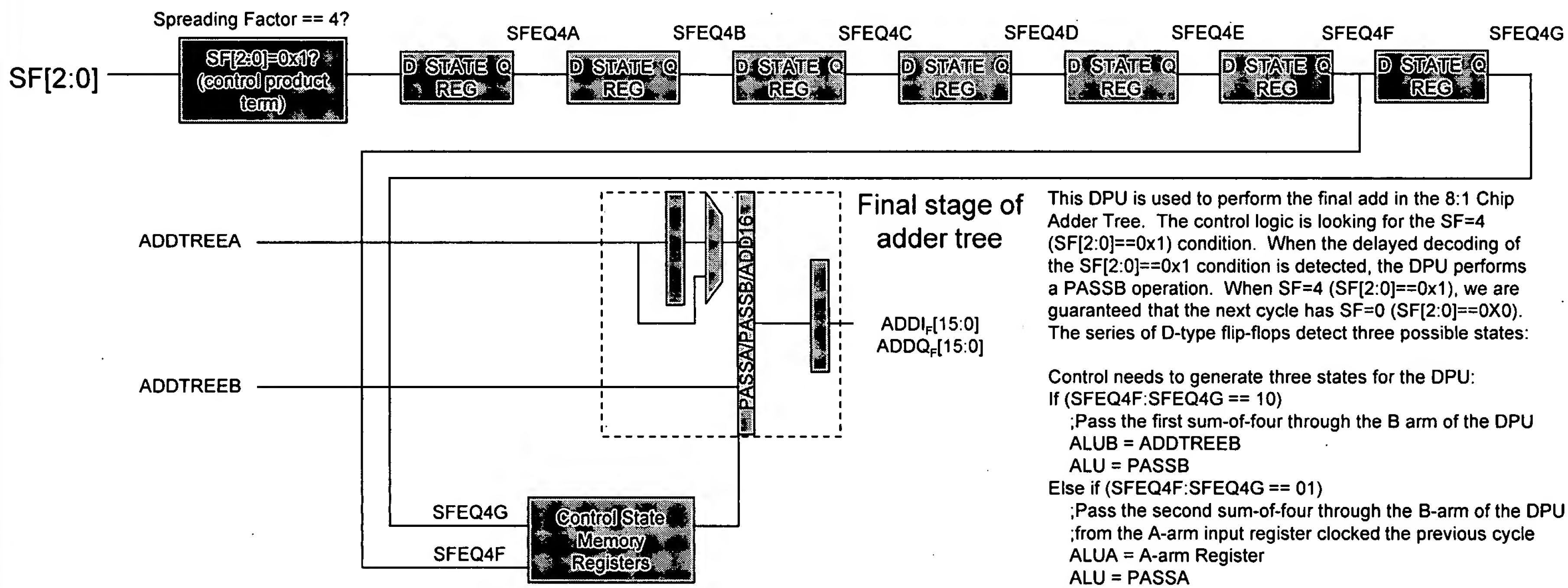
Chip Adder Tree Implementation

Outputs from Chip Descrambler Input Alignment / Chip Multiplier Circuit



Chip Adder Tree Control Implementation

We need delayed versions of the Spreading Factor (SF) to control the Adder Tree



This DPU is used to perform the final add in the 8:1 Chip Adder Tree. The control logic is looking for the SF=4 (SF[2:0]==0x1) condition. When the delayed decoding of the SF[2:0]==0x1 condition is detected, the DPU performs a PASSB operation. When SF=4 (SF[2:0]==0x1), we are guaranteed that the next cycle has SF=0 (SF[2:0]==0X0). The series of D-type flip-flops detect three possible states:

Control needs to generate three states for the DPU:

```
If (SFEQ4F:SFEQ4G == 10)
   ;Pass the first sum-of-four through the B arm of the DPU
   ALUB = ADDTREEB
   ALU = PASSB
Else if (SFEQ4F:SFEQ4G == 01)
   ;Pass the second sum-of-four through the B-arm of the DPU
   ;from the A-arm input register clocked the previous cycle
   ALUA = A-arm Register
   ALU = PASSA
Else
   ALUA = ADDTREEA
   ALUB = ADDTREEB
   ALU = ADD16
```


CHAMELEON SYSTEMS, INC.

Chip Adder Tree Control Timing

Fn = Data for Finger n is valid

SF RAM Outputs	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18	F19
UCB[2:0]=SF[2:0] valid in PLA	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18
Descrambler ALIGN Output	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
Descrambler CMULT Output	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
Adder Tree Final Stage Inputs	F248	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
SFEQ4A=Reg. Decode of SF=4	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
SFEQ4B=Reg decode of SFEQ4A	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
SFEQ4C=Reg decode of SFEQ4B	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
SFEQ4D=Reg decode of SFEQ4C	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14
SFEQ4E=Reg decode of SFEQ4D	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
SFEQ4F=Reg decode of SFEQ4E	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
Final Stage DPU CSR Inputs	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
Final Stage ALU Control Inputs	F248	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
SFEQ4G=Reg decode of SFEQ4F	F248	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11

Chip Adder Tree Resource Requirements

- Implementation of:

  32 Users @ 125 MHz

  48 Users @ 187.5 MHz

  64 Users @ 250 MHz

  - 7 DPUs
  - 0 LSMs

Chip Integrator Requirements and Assumptions

- Requirements
  - Sum SF (Spreading Factor) partial sums into a single sum of SF chips
  - Prepare Data that is to be sent to the Channel Estimator in the ARC
- Assumptions
  - Average SF = 128
  - 256 chips input rate per 256 clocks @ 125 MHz

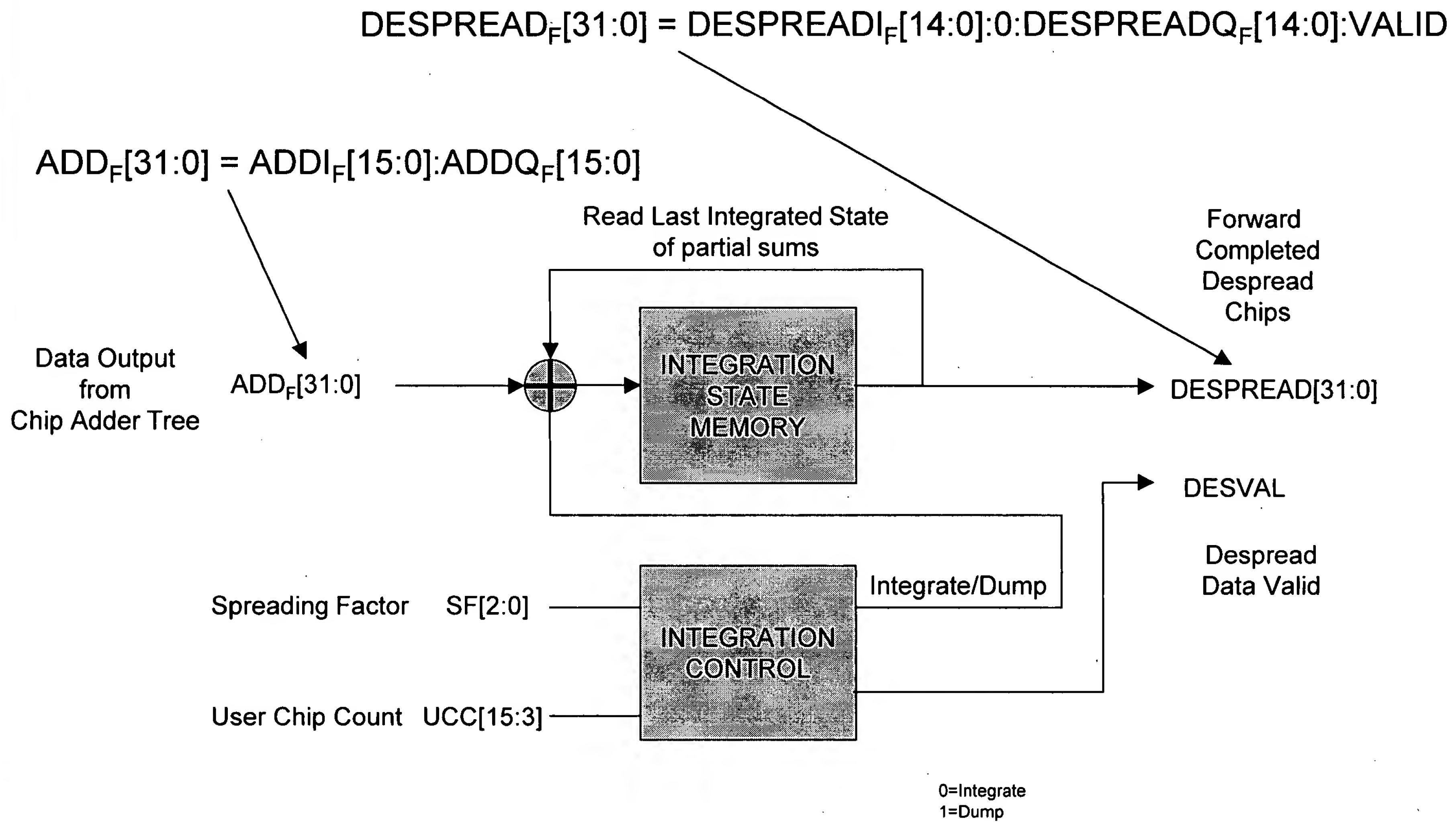
Chip Integrator
Theory of Operation

- Read one complex sum from Chip Adder Tree every clock cycle
- The LSB position of the Q component of the accumulated sum will be used to hold a valid bit for the backend circuits
- The LSB of the sum with SF=4 will have this LSB "robbed" without a noticeable effect on the result
- The accumulated sums with SF=4-128 will have their full-precision results shifted up one bit position
- For data with SF=4-128, shift input data up one bit position, for data with SF=256, pass data straight through
- Input data has already been despread by 8 chips (4 chips for fingers with SF=4)
- Sum input data with partial sum until SF chips (SF/8 inputs) have been added together

Chip Integrator
Theory of Operation

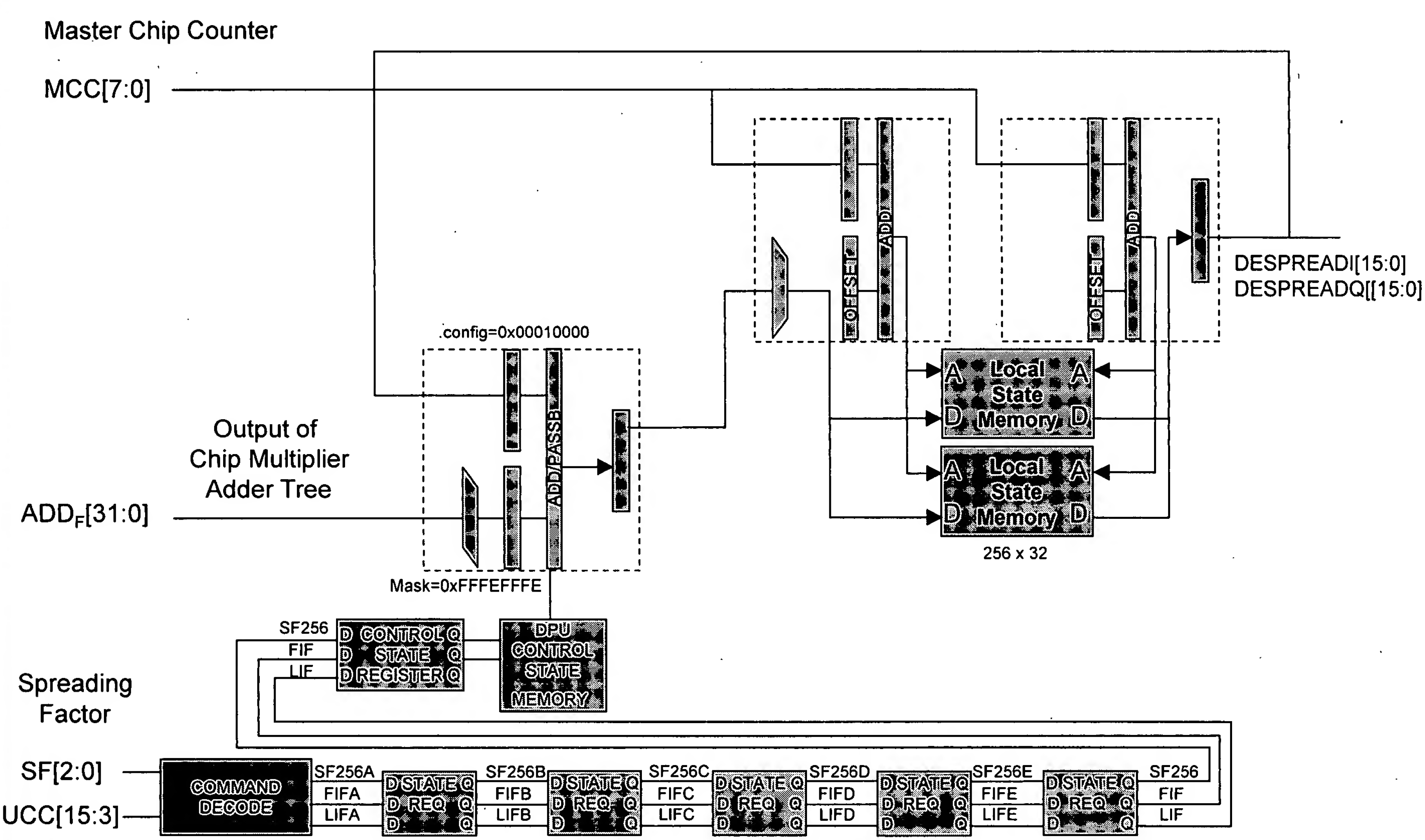
- If the Chip Integrator input has SF=4-128 shift the input data up one bit position and set bits 0 and 16 to zero
- If the Chip Integrator input is the last 8-chip sample in a set of SF/8 chips, add it to the partial sum and place it into Chip Integrator Memory and set the valid bit (bit 0)
- If the Chip Integrator input is the first 8-chip sample in a set of SF/8 chips, place it into the Chip Integrator Memory and forward the previous despread sum

Chip Integrator Functional Block Diagram



CHAMELEON SYSTEMS, INC.

Chip Integrator Datapath Implementation



Pipeline delays to align the control with the data

Chip Integrator Control Implementation (1/2)

```
First In Frame FIF =  SF[2:0]==0                              ;Spreading Factor = 0
                    + SF[2:0]==1                              ;Spreading Factor = 4
                    + SF[2:0]==2                              ;Spreading Factor = 8
                    + (SF[2:0]==3 && UCC[3]== 0x0)            ;Spreading Factor = 16
                    + (SF[2:0]==4 && UCC[4:3]== 0x0)          ;Spreading Factor = 32
                    + (SF[2:0]==5 && UCC[5:3]== 0x0)          ;Spreading Factor = 64
                    + (SF[2:0]==6 && UCC[6:3]== 0x0)          ;Spreading Factor = 128
                    + (SF[2:0]==7 && UCC[7:3]==0x00)          ;Spreading Factor = 256
```

```
Last In Frame LIF =   SF[2:0]==0                              ;Spreading Factor = 0
                    + SF[2:0]==1                              ;Spreading Factor = 4
                    + SF[2:0]==2                              ;Spreading Factor = 8
                    + (SF[2:0]==3 && UCC[3]== 0x1)            ;Spreading Factor = 16
                    + (SF[2:0]==4 && UCC[4:3]== 0x3)          ;Spreading Factor = 32
                    + (SF[2:0]==5 && UCC[5:3]== 0x7)          ;Spreading Factor = 64
                    + (SF[2:0]==6 && UCC[6:3]== 0xF)          ;Spreading Factor = 128
                    + (SF[2:0]==7 && UCC[7:3]==0x1F)          ;Spreading Factor = 256
```

```
Spreading Factor Equals 256 SF256 = SF[2:0]==7                ;Spreading Factor = 256
```

Chip Integrator Control Implementation (1/2)

```
CASE (SF256:FIF:LIF)
 000: ;Add 8-chip input to memory contents
      ALUA= AINPUT && Mask=0xFFFEFFFE
      ALUB= (BINPUT << 1) &&& Mask=0xFFFEFFFE
      ALU= ALUA + ALUB
 001: ;Add 8-chip input to Chip Integrator Memory contents and set VALID bit
      ALUA= AINPUT && Mask=0xFFFEFFFE
      ALUB= (BINPUT << 1) &&& Mask=0xFFFEFFFE
      ALU= ALUA + ALUB + 1 ;Set VALID bit
 010: ;Store 8-chip input into Chip Integrator Memory and forward previously despread sum
      ALUA= AINPUT && Mask=0xFFFEFFFE
      ALUB= (BINPUT << 1) &&& Mask=0xFFFEFFFE
      ALU= PASSB
 011: ;Add 8-chip input to Chip Integrator Memory contents, set VALID bit,
      ;and forward previously despread sum
      ALUA= AINPUT && Mask=0xFFFEFFFE
      ALUB= (BINPUT << 1) &&& Mask=0xFFFEFFFE
      ALU= ALUA + ALUB + 1 ;Set VALID bit
```

Chip Integrator Control Implementation (2/2)

```
CASE (SF256:FIF:LIF) (CONTINUED)
 100: ;Add 8-chip input to memory contents
      ALUA= AINPUT
      ALUB= BINPUT
      ALU= ALUA + ALUB
 101: ;Add 8-chip input to Chip Integrator Memory contents and set VALID bit
      ALUA= AINPUT && Mask=0xFFFEFFFE
      ALUB= BINPUT &&& Mask=0xFFFEFFFE
      ALU= ALUA + ALUB + 1 ;Set VALID bit
 110: ;Store 8-chip input into Chip Integrator Memory and forward previously despread sum
      ALUA= AINPUT
      ALUB= BINPUT
      ALU= PASSB
 111: ;THIS INPUT COMBINATION IS NOT POSSIBLE
      ;ARBITRARILY ASSIGN THE SAME COMMAND AS CASE 110
      ALUA= AINPUT
      ALUB= BINPUT
      ALU= PASSB
```

00220T" h6986960

Chip Integrator Control Timing

Fn = Data for Finger n is valid

SF/UCC RAM Outputs	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18	F19
SF/UCC valid in PLA	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18
Descrambler ALIGN Output	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
Descrambler CMULT Output	F254	U62A	U62B	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
Adder Tree Final Stage Outputs	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
INTA=Reg Decode of SF/UCC	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
INTB= Reg decode of INTA	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16
INTC= Reg decode of INTB	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
INTD= Reg decode of INTC	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14
INTE= Reg decode of INTD	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
DataValid CSR Inputs	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14
DataValid ALU Inputs	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
DataValid Output Register Valid	F249	F250	F251	F252	F253	F254	F255	F0	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
Integrate/Dump Output Reg Val	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
Integrate/Dump CSR Inputs	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
Integrate/Dump ALU Inputs	F249	F250	F251	F252	F253	F254	F255	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12

Chip Integrator Resource Requirements

- Implementation of:

  32 Users @ 125 MHz

  48 Users @ 187.5 MHz

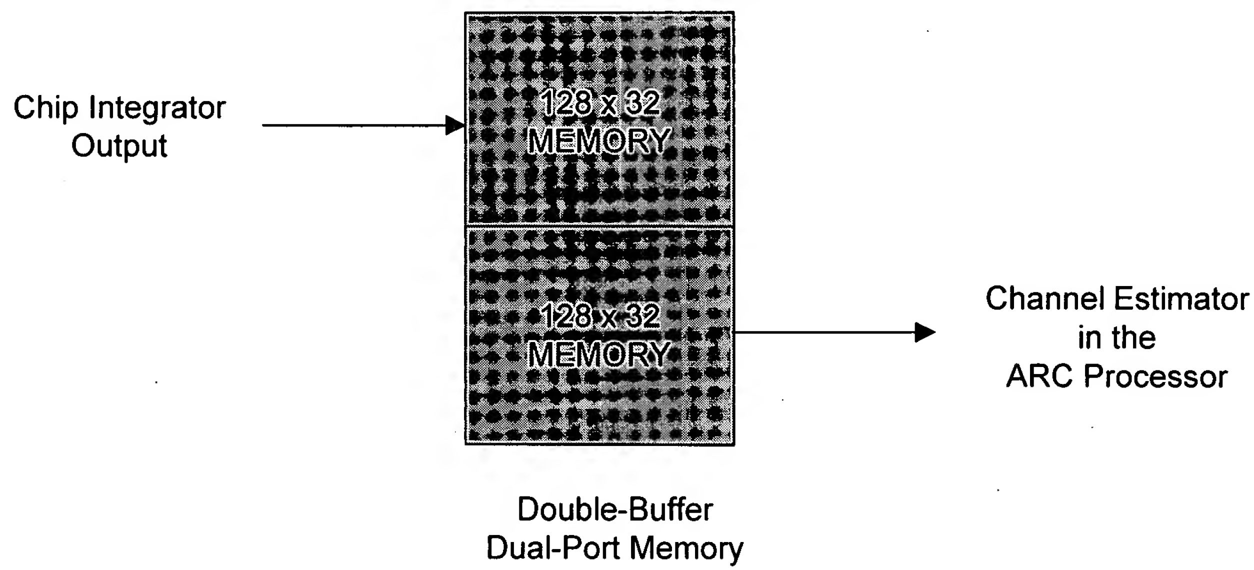
  64 Users @ 250 MHz

  - 3 DPUs
  - 2 LSMs

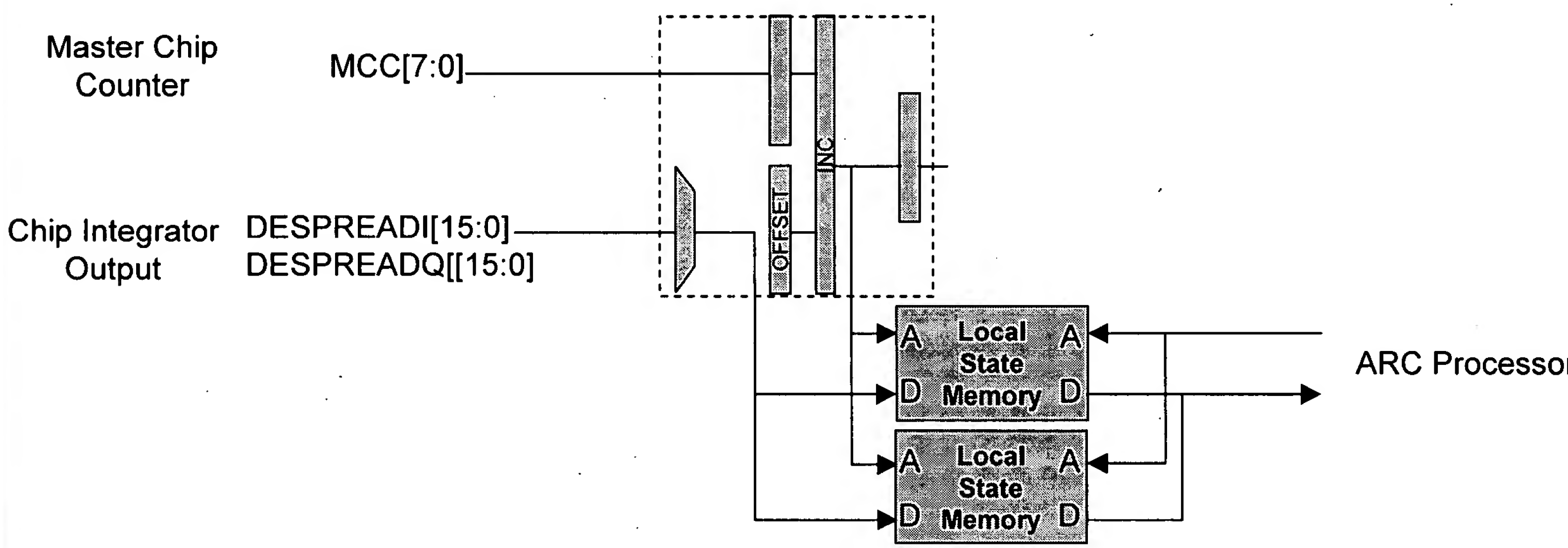
Channel Estimator Data Input Buffer Requirements and Assumptions

- Requirements
  - Provide a path between the Chip Integrator in the Chameleon fabric and the ARC processor
  - Double-buffer 128 despread pilot symbols every 256 T_c
- Assumptions
  - Input written by Chip Accumulator
  - Output is read by the ARC core
  - All Control Channel data has SF=256

Channel Estimator Data Input Buffer Functional Block Diagram



Channel Estimator Data Input Buffer Implementation



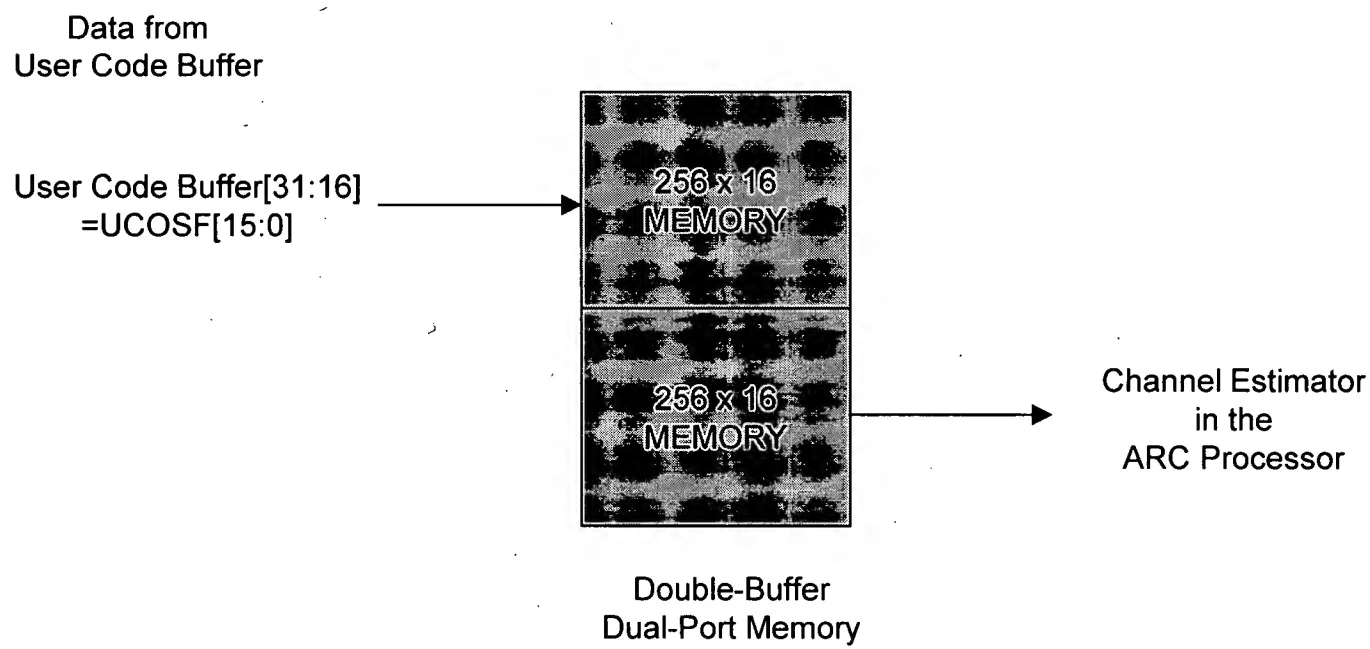
Channel Estimator Data Input Buffer Resource Requirements

- 32 User Implementation @ 125 MHz
  - 1 DPU
  - 2 LSMs
- 48 User Implementation @ 187.5 MHz
  - 1 DPUs
  - 4 LSMs
- 64 User Implementation @ 250 MHz
  - 1 DPUs
  - 6 LSMs

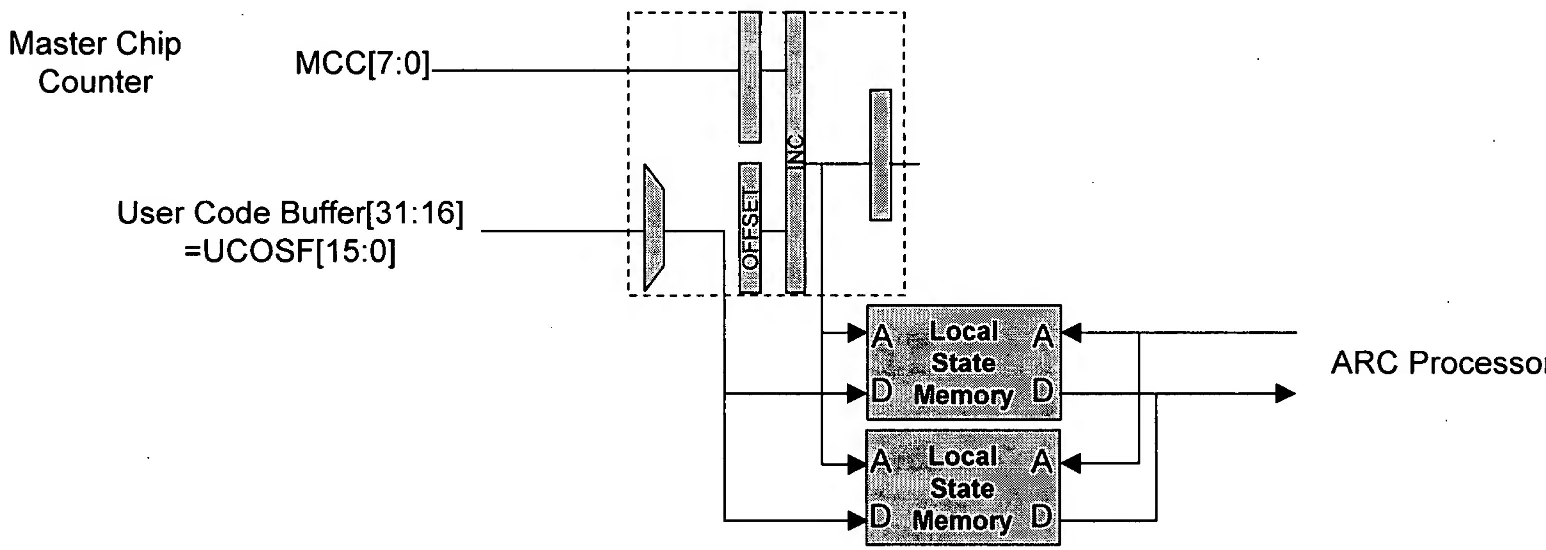
Channel Estimator UCC Input Buffer Requirements and Assumptions

- Requirements
  - Provide a buffer containing the present User Chip Counter (UCC) value for each of the 128 Pilot chips
  - Provide a buffer containing the present User Chip Counter (UCC) value for each of the 128 Data chips
  - Double-buffer 256 UCCs every 256 T_c
- Assumptions
  - Input written by User Code Buffer
  - Output is read by the ARC core

Channel Estimator UCC Input Buffer Functional Block Diagram



Channel Estimator UCC Input Buffer Implementation



Channel Estimator UCC Input Buffer Resource Requirements

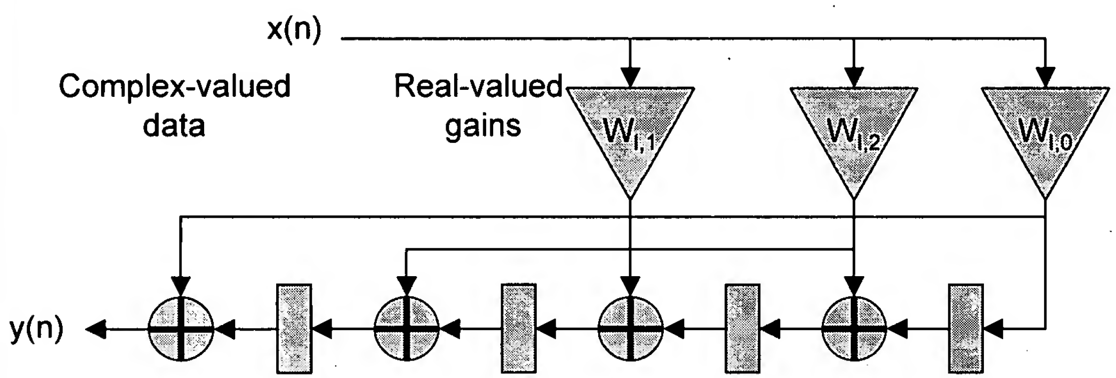
- 32 User Implementation @ 125 MHz
  - 1 DPU
  - 2 LSMs
- 48 User Implementation @ 187.5 MHz
  - 1 DPUs
  - 4 LSMs
- 64 User Implementation @ 250 MHz
  - 1 DPUs
  - 6 LSMs

Channel Estimator Requirements and Assumptions

- Requirements
  - Compensate the user data given the characteristics of the Pilot Channel data using XXX filtering
- Assumptions
  - All channel estimation is performed in the ARC processor
  - The Channel estimation is performed after the Pilot Channel has been despread (SF=256) and is significantly slower than the chip rate

FIR Filters for Channel Estimation

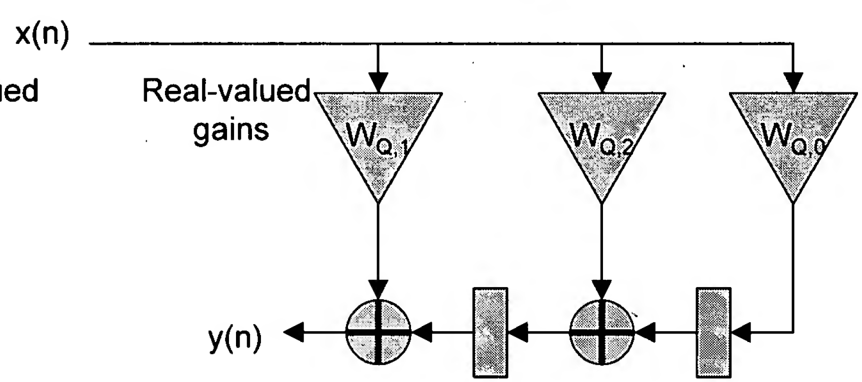
5-Tap Symmetric FIR Filter



Required Ops/Sample:
- 6 Multiplies
- 4 Additions
- 3 Post-Multiply Packing Ops

Allow Multiple Clocks per Sample:
- 2 MUL for multiplies
- 1 DPU for additions
- 1 DPU for packing

3-Tap Non-symmetric FIR Filter



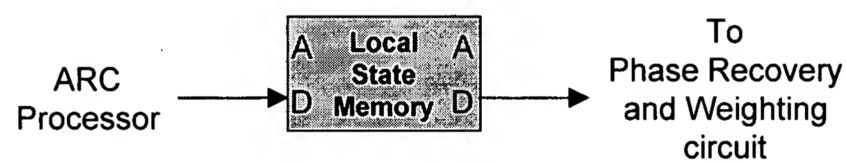
Required Ops/Sample:
- 6 Multiplies
- 2 Additions
- 3 Post-Multiply Packing Ops

Allow Multiple Clocks per Sample:
- 2 MUL for multiplies
- 1 DPU for additions
- 1 DPU for packing

Channel Estimator Output Buffer Requirements and Assumptions

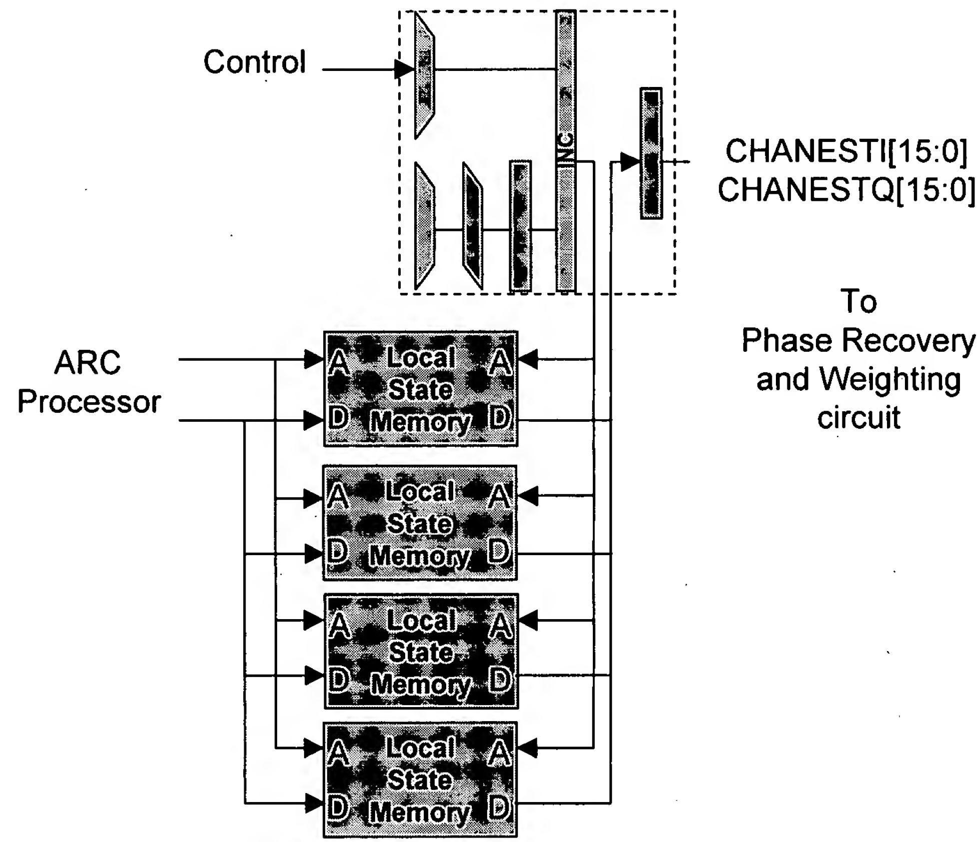
- Requirements
  - Provide a buffer between the Channel Estimation Filter and the Phase Recovery circuits
  - Provide a double buffer memory to prevent race conditions
- Assumptions
  - One 32-bit (16-bit I, 16-bit Q) Channel Compensation Weight word per pilot is required
  - 32 Users required
  - Must provide double buffer for proper operation
  - Channel Compensation word is sample and held on one time-slot (2560 T_c) boundaries

Channel Estimator Output Buffer Functional Block Diagram



256 Fingers: 256 fingers x 32-bit words x 2 banks (ping, pong)

Channel Estimator Output Buffer Implementation



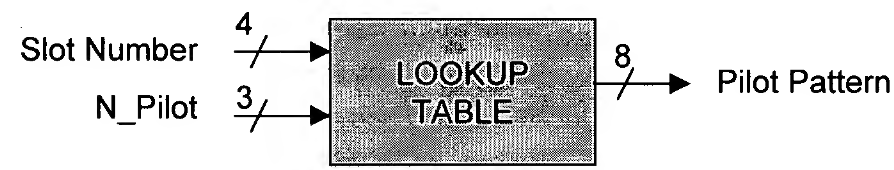
Channel Estimator Output Buffer Resource Requirements

- 32 Users Implementation
  - 1 DPU
  - 2 LSMs
- 48 Users Implementation
  - 2 DPU
  - 3 LSMs
- 64 Users Implementation
  - 2 DPU
  - 4 LSMs

Pilot Pattern Generator Requirements and Assumptions

- Requirements
  - Provide simple lookup-table approach
- Assumptions
  - Physically resides in ARC processor
  - N_Pilot = 3,4,5,6,7, or 8
  - The pilot is a function of slot
  - A single LSM is sufficient to contain the lookup-table

Pilot Pattern Generator Functional Block Diagram



Pilot Pattern Generator Memory Map Address Bit Definitions

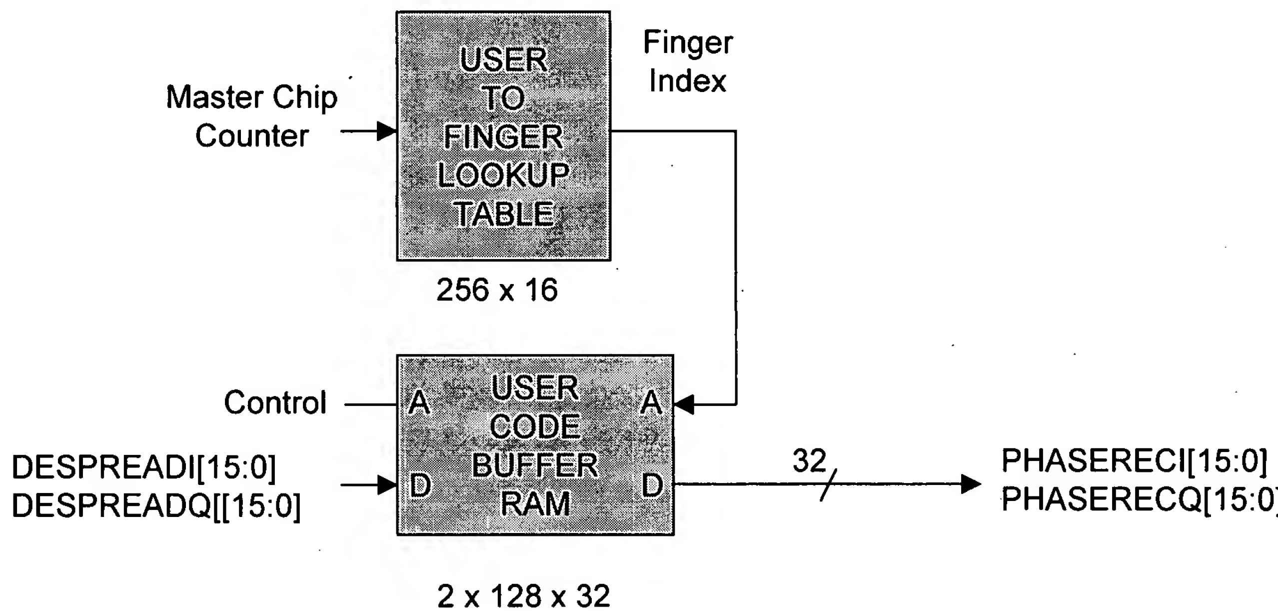
N_Pilot	PilotGenAddr[6:4]
3	0
4	1
5	2
6	3
7	4
8	5

SlotNumber	PilotGenAddr[3:0]
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14

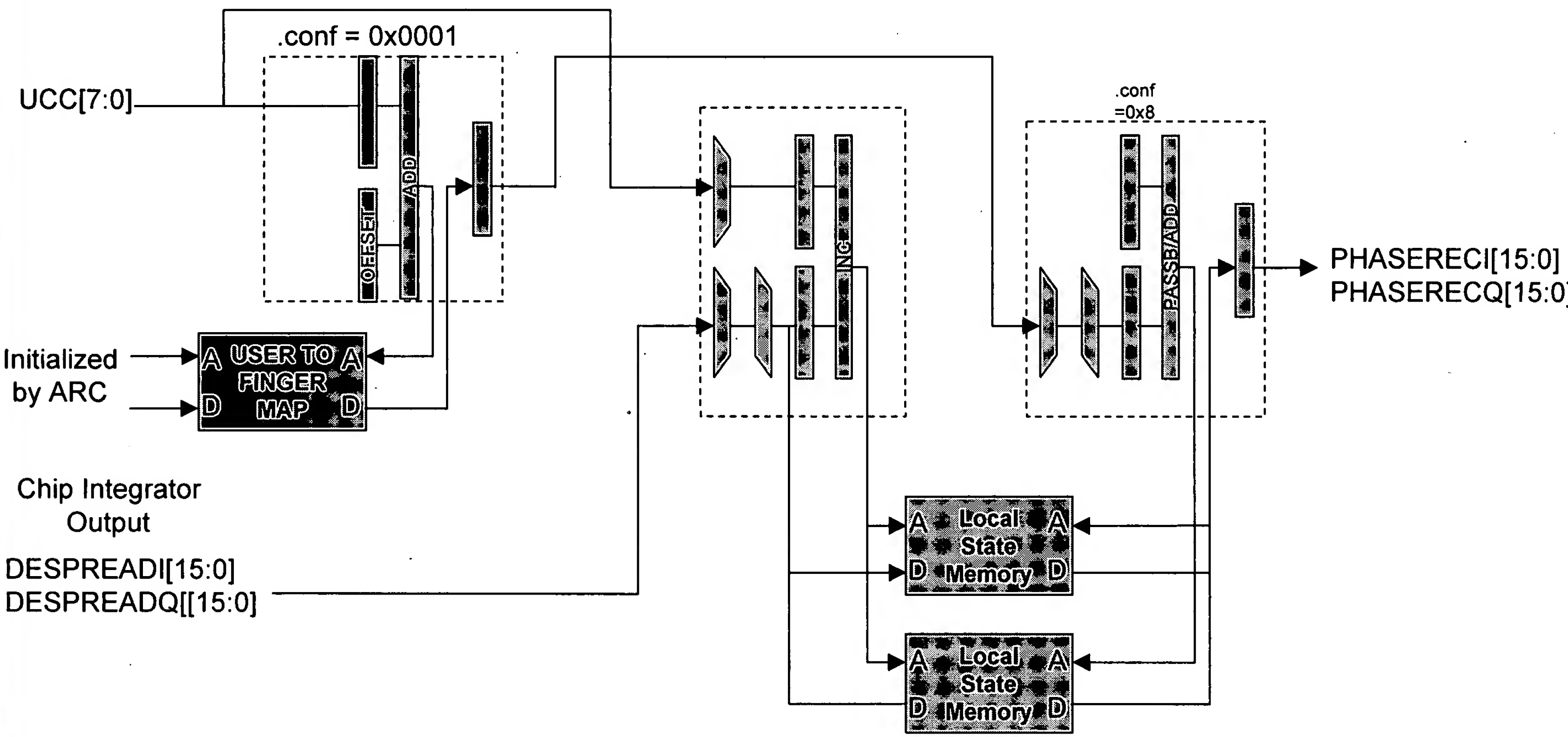
Phase Recovery Input Buffer Requirements and Assumptions

- Requirements
  - Provide a double buffer between the Chip Integrator circuit and the Phase Recovery circuit
  - Buffer 128 fingers every 256 clocks
- Assumptions
  - Only the delayed data channels are buffered
  - Pilot Channel data is not buffered

Phase Recovery Input Buffer Block Diagram



Phase Recovery Input Buffer Implementation



Phase Recovery and Weighting Requirements and Assumptions

- Requirements
- Assumptions
  - A complex multiplication of (A +jB) * (C +jD)

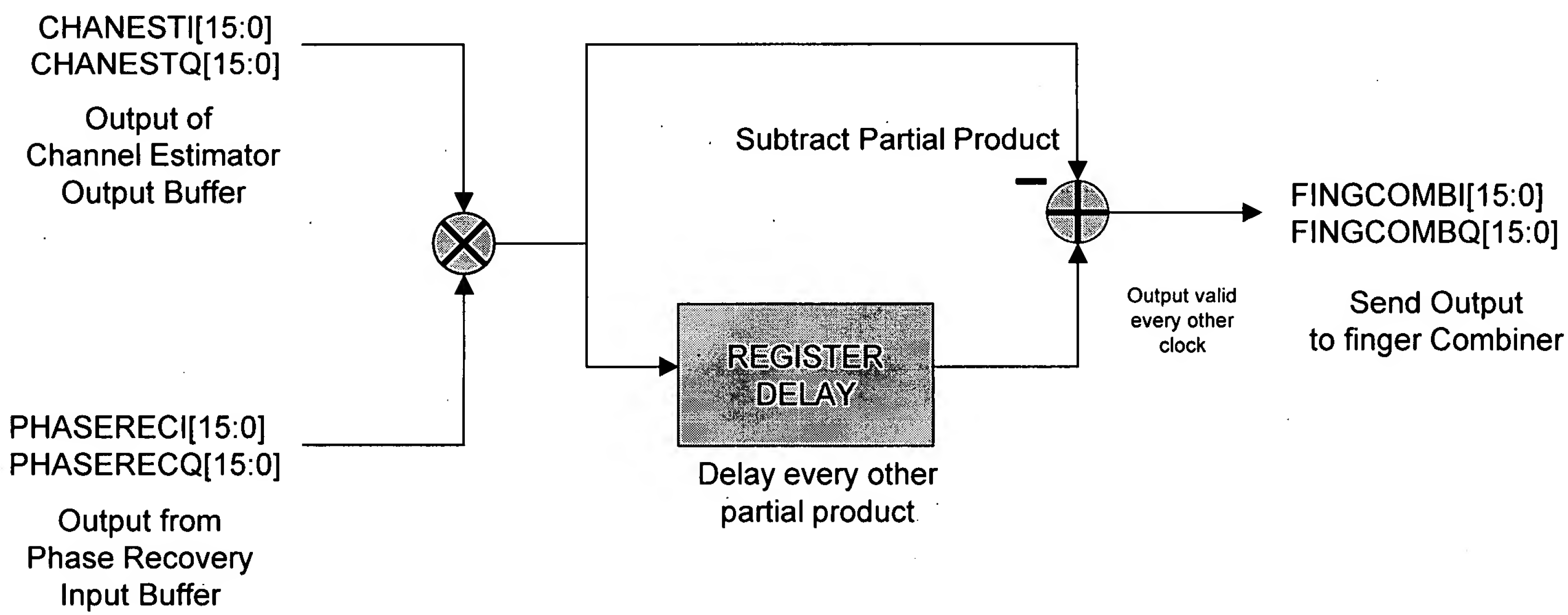
    $$= (AC - BD) + j(AD + BC)$$

  - We are only interested in the real part of the output:

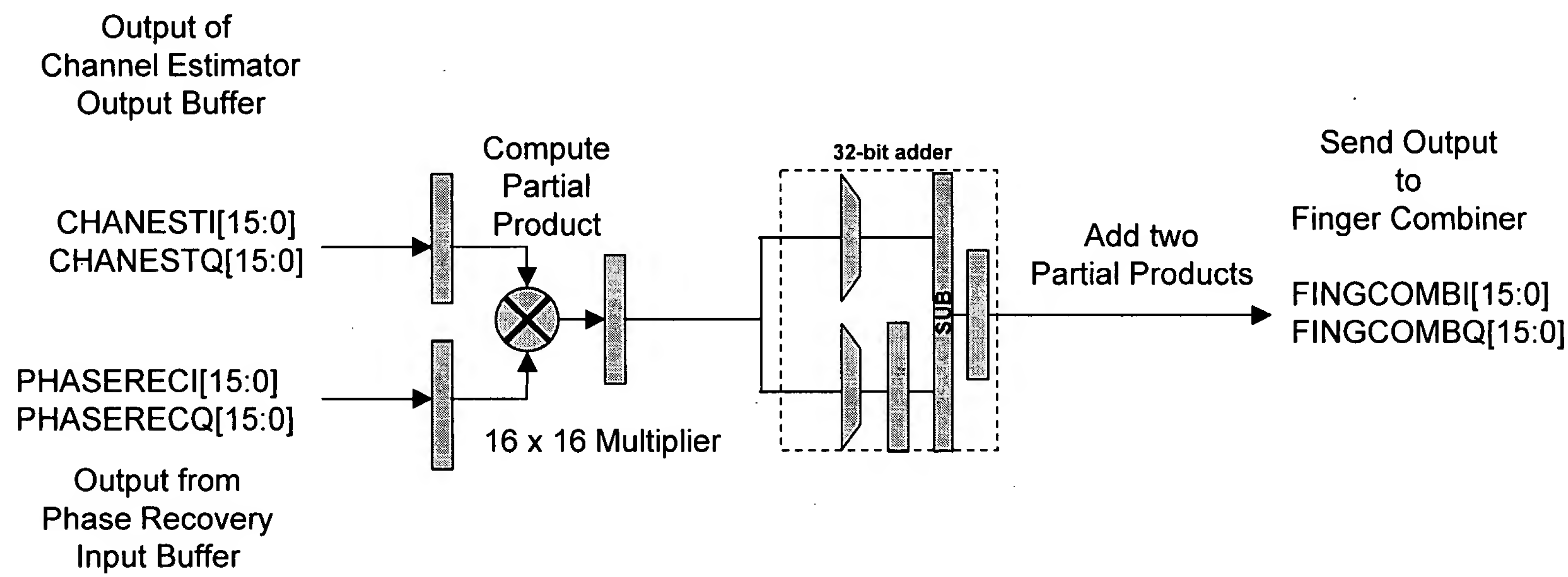
    $$Re\{(A + jB) * (C + jD)\} = AC - BD$$

  We have extra cycles so only one multiplier is required

Phase Recovery and Weighting Functional Block Diagram



Phase Recovery and Weighting Implementation



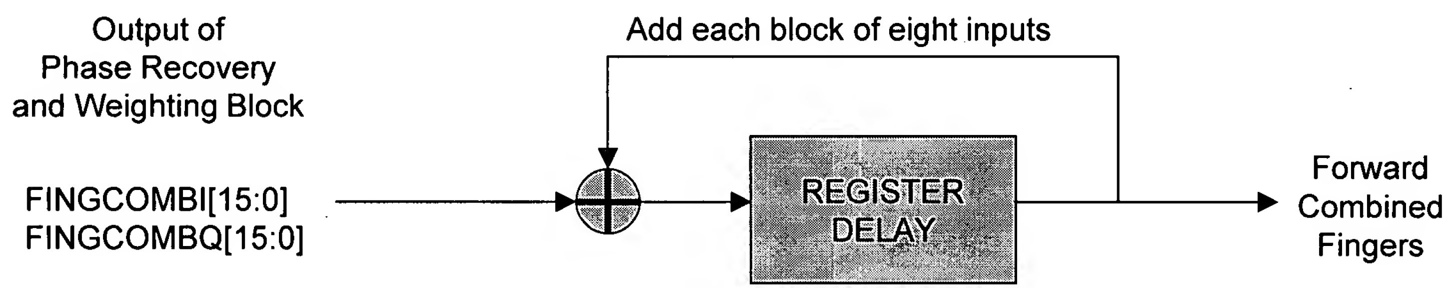
Phase Recovery and Weighting Resource Requirements

- Implementation of:
  32 Users @ 125 MHz
  48 Users @ 187.5 MHz
  64 Users @ 250 MHz
  - 1 DPU
  - 1 Multiplier

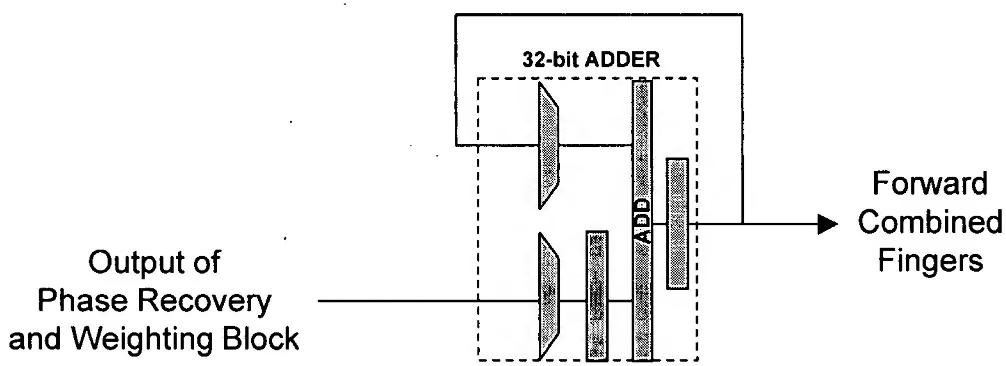
Finger Combiner
Requirements and Assumptions

- Requirements
  - Must be able to combine up to six fingers
- Assumptions
  - Extra cycles can be wasted as long as the TPCG < 300us
  - Allocate timing such that the circuit is always adding 8 fingers
  - The ARC processor may assign up to 8 fingers to each user
  - The ARC processor assigns zeroes to the unused fingers in an eight finger block

Finger Combiner Functional Block Diagram



Finger Combiner Implementation



Finger Combiner Resource Requirements

- Implementation of:

  32 Users @ 125 MHz

  48 Users @ 187.5 MHz

  64 Users @ 250 MHz

  - 1 DPU

General Timing and Control
Finger Tracking within the Rake Receiver

- Upon initial acquisition, the Rake Receiver is tasked such that the Scrambling Code is set to an offset that places the (up to) six fingers in the Antenna Sample Buffer window such that zero path delay corresponds to the beginning of the buffer window
- As the mobile user moves toward or away from the base station (Node B), the fingers will move within the Antenna Sample Buffer window as tasked by the Path Searcher

Spreading Factor Mapping Assignments

- The following table lists the values corresponding to the various Spreading Factors (SF):
- Note that users requiring SF=4 must assign each finger to two successive finger resources per finger according to the following rules
  - At finger resource n assign SF=4
  - At finger resource n+1 assign SF=0 (used to denote second part of SF=4 finger)

SPREADING FACTOR	SF MEMORY CONTENTS
0	0
4	1
8	2
16	3
32	4
64	5
128	6
256	7

Chameleon Systems Rake Receiver CS2112 Device Utilization for 32 Users

KERNEL	DATA PATH UNITS (DPUs)	LOCAL STATE MEMORIES (LSMs)	MULTIPLIERS
Gold Code Generator	14	8	2
Channel Code Generator / Multiplier	4	2	0
Antenna Sample Buffer	22	26	0
Chip Descrambler	24	0	0
Chip Adder Tree	7	0	0
Chip Integrator	3	2	0
Channel Estimator Data Input Buffer	1	2	0
Channel Estimator UCC Input Buffer	1	2	0
Channel Estimator Output Buffer	1	2	0
Phase Recovery Input Buffer	3	3	0
Phase Recovery and Weighting	1	0	1
Finger Combiner	1	0	0
Master Chip Counter	2	0	0
TOTALS:	84	47	3
TOTAL AVAILABLE	84	48	24
PERCENT UTILIZED	100%	98%	13%